ARPA ORDER NO.: 189-1
7P10 Information Processing Techniques

R-1808-ARPA
September 1977

RITA Reference Manual

R. H. Anderson, Margaret Gallegos,
J. J. Gillogly, R. Greenberg, R. Villanueva

A report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under Contract No. DAHC15-73-C-0181.

Published by The Rand Corporation

ARPA ORDER NO.: 189-1
7P10 Information Processing Techniques

R-1808-ARPA
September 1977

RITA Reference Manual

R. H. Anderson, Margaret Gallegos,
J. J. Gillogly, R. Greenberg, R. Villanueva

A report prepared for

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

PREFACE

Members of Rand's computer science research program are currently implementing a set of intelligent terminal agent computer programs called RITA. This effort is part of a larger research program on advanced intelligent terminals being funded and coordinated by the Information Processing Techniques Office (IPTO) of the Defense Advanced Research Projects Agency (ARPA) under the program management of Stephen Walker.

This report—one of a series documenting the RITA system—provides a detailed reference manual for RITA users, including a description of the RITA language and operating environment. It is intended primarily as a tool for users who will be developing applications using RITA, not as a discussion of implementation issues or design philosophy or as a tutorial document for beginning users.

The RITA system is designed to be widely applicable as a front end to remote computing systems and networks, and as a limited heuristic modeling tool. Thus, this reference manual should be of interest to persons involved with the design of interfaces to computer networks for logistics, maintenance scheduling and control, and command and control systems; for intelligence collection and dissemination; and for remote accessing of large data bases. It should also be useful to designers of minicomputer software systems supporting administrative and command functions.

This report should be read in conjunction with a companion Rand report in the series, *Rand Intelligent Terminal Agent (RITA): Design Philosophy,* R-1809-ARPA, February 1976, which presents the rationale for intelligent terminal agent programs and for the use of production system techniques to implement such agents.

SUMMARY

The RITA system is a set of computer programs written in the C programming language that can run under the UNIX operating system on minicomputers such as the PDP 11/45 and PDP 11/70. With RITA, the user can rapidly develop "user agents" to perform such tasks as providing a simple interface to remote data systems; the system also provides some heuristic modeling capability.

RITA makes available a language for writing rules and an operating environment in which those rules are interpreted; this allows the user to construct a set of IF-THEN rules which can operate both on a local data base consisting of object-attribute-value triples and on data received over communication links to external systems. The major portion of this reference manual is an alphabetical listing of RITA concepts, keywords, and commands. A complete syntax chart for the system, with tables of reserved words and built-in functions, is included in a pocket on the inside back cover.

CONTENTS

I. INTRODUCTION

The RITA system is a set of computer programs designed to allow the efficient development of "user agents" that can reside in an intelligent terminal. Minicomputers such as the PDP 11/45 and PDP 11/70 are currently being used as a surrogate for the computing power and memory which will soon reside physically within a user's terminal.

The RITA programs are written in the C language and run under the UNIX operating system. All user agents written using the RITA system consist of a set of pattern-replacement rules expressed in a simple English-like language.

The design philosophy underlying RITA is discussed in a companion report, *Rand Intelligent Terminal Agent (RITA): Design Philosophy*, R-1809-ARPA, February 1976; we strongly recommend that the user read this companion report before undertaking a serious development effort involving RITA.

This reference manual is a concise digest of technical information sufficient to allow the user to create RITA rule sets. A RITA rule set is the RITA equivalent of a program and is sometimes referred to herein as a user agent. This manual is not written as a tutorial document, but rather as an encyclopedia of necessary facts. The major portion of the manual consists of an alphabetical listing of concepts, keywords, and commands available in the RITA system. In addition, concise tabular summaries of RITA rule syntax and lists of reserved words and built-in functions are provided in App. A, which is located in a pocket on the inside back cover.

II. OVERVIEW

The RITA system can be thought of as providing a basic starter set of commands and options for the creation of rule sets. Also included are various extensions in such specialized areas as sending and receiving streams of characters to and from external information systems. Table 1 shows the commands that can be issued to the RITA system and the keywords used to construct goals and rules. The commands and keywords are organized into logical groups to aid the beginning user in accessing concepts within the alphabetical listing in Sec. III.

We suggest that the beginning user start by reading the following entries in Sec. III:

- rita
- monitor
- object
- attribute
- data type
- value
- rule

He should then access the entries for the commands and keywords for "basic" RITA listed in Table 1 and gradually add subsets of commands and keywords as needed.

A RITA program is a set of commands or directives which the RITA system can perform, together with a set of rules which can be tested and whose actions may be performed if the premises tested are true. RITA's "data" are kept in the form of objects that can be created either through directives or through the actions of rules. A RITA program is sometimes called a *RITA rule set*; it may be referred to as a *user agent* when it is used to perform an interactive task with another user or with a computer system.

We envision RITA as a tool that will allow non-computer specialists to modify and even create simple user agents to help handle routine interaction with local and remote information systems. RITA can also be used to develop heuristic models to assist in decisionmaking and analysis. (This capability is not discussed in detail in this manual but will be more thoroughly described in forthcoming reports.) Heuristic models are most often deductive in nature, so we have provided an example of a deductive rule set in the wall chart (App. A).

The version of RITA described herein might well be the first of several design iterations. We have intentionally kept both the language and the data structures very simple, in order to give RITA an English-like readability, and we have tried to avoid recursiveness and generality. Before proceeding with the next level of extensions and refinements to this evolving prototype, we wanted to give new users the opportunity to provide feedback. This manual is being released to provide such an opportunity and to encourage user participation in identifying areas for further design enhancement.

The remainder of this overview will describe the context in which RITA is used and in which user agents are developed and used. Appendix B contains an example

Table 1

RITA Commands and Keywords

Category	Commands	Keywords[a]
"Basic" RITA	LOAD RUN EXIT	RULE OBJECT IF, THEN THERE IS . . . WHOSE SET . . . TO IS [NOT] KNOWN IS [NOT] IS [NOT] [GREATER THAN] IS [GREATER THAN] IS [NOT] [LESS THAN] RETURN [SUCCESS][FAILURE] CREATE DELETE DELAY
Input/output extensions		SEND . . . TO CONTAINS DOES NOT CONTAIN RECEIVE . . . FROM RECEIVE NEXT pattern specification
List manipulation extensions		IS [NOT] IN PUT . . . INTO FIRST, LAST, AFTER MEMBER REMOVE . . . FROM SIZE OF
Tracing and debugging	TRACE, UNTRACE STOP AT UNSTOP DISPLAY	TRACE, UNTRACE STOP AT UNSTOP DISPLAY . . . [TO FILE] ALL . . . THAT
Goal-directed operation		GOAL DEDUCE
Explanation subsystem	WHY WHAT	
Misc. utility commands	⟨CTRL FS⟩ SHELL EDIT QUIET, VERBOSE NEWS SCRATCH	

[a]Brackets indicate optional variations.

of a complete agent, as well as some discussion of the interactive task being performed by the agent and the system with which it interacts.

OPERATING ENVIRONMENT

RITA runs under the UNIX operating system and makes use of many capabilities provided by that system. These will be referred to occasionally to allow the reader to differentiate between RITA and its environment. RITA, when installed on a UNIX system, can be accessed via the command *rita.*

RITA is normally used in an interactive mode during the creation of a rule set, or agent. That is, the user normally invokes RITA from an on-line terminal connected to the UNIX system and then proceeds to enter rules and create objects. During this phase, RITA checks for syntax errors and stores named rules (and goals, which are special kinds of rules used in deductions), rather than acting on them or testing them. The user may also use directives or commands during this initial development phase to request RITA to do such things as display the currently stored rules or objects for review. Immediate rules (rules which are not named and therefore not stored) may also be tested and their actions executed if appropriate. Throughout this interaction, RITA prompts the user for more input by sending an asterisk (*) to the terminal. The user terminates each input to RITA with a semicolon (;) to indicate that RITA can begin to process that input.

To test entered rules and goals, the user types the *run* command, which causes the selected or default monitor to begin testing and executing rules whose premises are true. At any time during a session with RITA, all the goals and rules may be stored in a UNIX file (by using the *display* command). Rules and goals so displayed are then available for loading by RITA either in the same session (perhaps after a *scratch* command) or in a later session.

Following UNIX convention, RITA can either take input from and interact with an on-line user at a terminal or take all of its input directly from a file. In the initial creation of a user agent, the user generally interacts with RITA via a terminal; while some agents will be designed to interact with the user as part of their function, others will be set free to work independently, taking their input and their rules from files and perhaps reporting to the user via messages or through files. Such agents will use the UNIX facility for redirecting input and output. For example, the command *rita* simply invokes RITA modules which take input from the user terminal on which they were called and presents responses (or output) onto the same terminal. The command *rita* < *inputfile* takes input from the file called *inputfile* and prints its responses on the terminal from which RITA was invoked. Similarly, the command *rita* < *inputfile* > *outputfile* takes input from the file called *inputfile* and sends output to the file called *outputfile.* The latter type of agent may be run in a "background" mode and need not tie up the user's terminal. In UNIX this is done simply by appending an ampersand (&) to the end of the command string as given above. Without the ampersand, UNIX will normally wait for the called process to be completed before returning the user to UNIX command level. The effect of this command is to return control immediately after RITA has been started.

When the command *rita* is given to UNIX at a terminal (UNIX command level is called the shell), RITA is invoked and prints a line telling the user how many

other RITA processes are currently running; this line is followed by three lines giving the dates and names of the three major RITA processes. (The dates identify the version that is in use.) These three lines are followed by an asterisk. RITA is now at command level and awaits input from the user. Example 1 shows the use of the *create* command. The session starts with the UNIX prompt, %, followed by the *rita* command and a short interaction between the user and RITA. Note that RITA prompts for input with an asterisk and the user terminates input with a semicolon followed by a carriage return.

Example 1

```
% rita

There are 56 processes in use (65% loading) and no other ritas
                                                        running

UFE: 1 Sep 76
PARSER: 2 Sep 76
MON: 1 Sep 76

* create a system whose name is "Unix";

* if there is a system whose name is not "Tenex"
then send "Eureka!" to the user;
Eureka!
RULE :
    IF: THERE IS a system<1> WHOSE name IS NOT "Tenex"
  THEN: SEND "Eureka!" TO user;

Success!

*
```

The *create* command creates a named data object within the RITA system. It is followed by an immediate rule. RITA tests the premise of the rule, finds it true, and executes the action. Since the *quiet* option was not used, RITA decompiles and prints the rule. It prints *Success!* to indicate the successful firing of an immediate rule and then prompts for more input.

This example was created at a terminal; therefore, the record of the interaction between RITA and the user is interleaved. The same user input is shown in Example 2, but this time it is provided in the form of a file. The file is called *example* and looks like this:

Example 2

```
create a system whose name is "Unix";


if there is a system whose name is not "Tenex"
then send "Eureka!" to the user;
```

Example 3 shows RITA output redirected to a file called *results*. The UNIX command used is *rita* < *example* > *results* &, which causes RITA to take all input from the file called *example* and send all output to the *results* file. *Results* then looks like this:

Example 3

```
There are 54 processes in use (63% loading) and no other ritas
                                                        running
UFE: 1 Sep 76
PARSER: 2 Sep 76
MON: 1 Sep 76


example:
Eureka!
RULE :
    IF: THERE IS a system<1> WHOSE name IS NOT "Tenex"
  THEN: SEND "Eureka!" TO user;



Success!


* exiting.
```

Notice that in the *results* file the input file name *example* has been added as the first line after the standard RITA header. This is followed by RITA responses to the input file commands; an asterisk appears to show that the system returned to command level after firing the rule successfully. At this point, RITA gets an end-of-file from its input (the *example* file) and prints the response *exiting.*. It is also possible to have RITA accept input from a file and then return to command level and accept input from the controlling terminal. In the above example, this would be done by giving the UNIX command *rita example.* The result of this command would be identical to that above, with two exceptions:

1. The output seen in the *results* file would be seen at the terminal.
2. Instead of the response *exiting.* after the last prompt, RITA would continue to take input from the primary input stream—in this case the terminal—until it received the command *exit;* or until it got an end-of-file from the terminal.

The *load* command would cause RITA to begin taking input from the named file until an end-of-file was reached and then to resume taking input from the primary input stream.

EXTERNAL SYSTEMS

The UNIX Operating System provides a pipe facility which RITA uses to open communications channels (ports) with the UNIX command level. The current version of RITA provides up to three ports to UNIX. These ports are each two-way communications paths and can be thought of as providing a RITA program with the ability to interact with UNIX in much the same way that a user would at a terminal. That is, the command level of UNIX is made available to RITA simply by sending a command string through a named port. Responses from the UNIX system will be waiting in a buffer associated with each port and may be read by receiving from that port. Since three such ports are available, three such concurrent interactions may be ongoing at once.

In order to make use of this feature directly, RITA users will have to know what is available on UNIX and how to use it. This information is given in the UNIX Programmer's Manual and the Documents for Use with the UNIX Time-Sharing System, which are available to UNIX facilities through Western Electric. RITA users who are familiar with remote systems that are available on the ARPAnet might use the port facility to open one or more connections to a more familiar environment and then use the RITA *send* and *receive* action clauses in conjunction with rules and goals to interact automatically with systems and subsystems whose expected range of responses they know. In the latter case, the RITA user needs to know only the proper protocol on UNIX for connection to a remote host via telnet in order to write agents to interact with such hosts or their subsystems.

For users who are unfamiliar with the ARPAnet and its log-in procedures, a successful script is shown in Example 4. (Comments are included in brackets.)

Example 4

```
% telnet rand-rcc                     [local host (unix) prompt]
                                      [ & user call to telnet  ]
                                      [ giving host.           ]

Connections established.              [telnet response         ]

                                      [one or more garbage or  ]
                                      [blank lines             ]

%%%%Ø%



                                      [remote-host herald      ]
                                      [line(s):                ]
RAND COMPUTATION CENTER    LINE 155  Ø3/Ø3/77   1:11:23 P.M.
THE SYSTEM WILL BE DOWN FROM 18ØØ-2ØØØ, TONIGHT.
USER? xØØØØ                           [prompts for user name,  ]
ACCOUNT? 2Ø4Ø2                        [account and keyword     ]
KEYWORD? ritar

COMMAND ?                             [remote host (rand rcc)  ]
                                      [prompt                  ]
```

Example 5 shows a file that contains rules and objects capable of making such a connection with a specific host. If the file's name were *telnet-agent*, the command *rita telnet-agent* would cause RITA to take input from the file and to store objects and rules as they are encountered in the input stream. Upon reading the *run* command, RITA cycles through the rules, testing and executing true rules. The comments given in the example are intended to clarify the rule set and point to entries in Sec. III that might provide pertinent information.

Example 5

```
object remote-system:                    [remote-system is an object  ]
        host-name is "rand-rcc",         [host-name is an attribute   ]
        login-prompt is "USER?",         [of an object.               ]
        prompt is "COMMAND ?",           [Objects can be created      ]
        exit-command is "logoff";        [simply by declaring their   ]
                                         [existence.                  ]
object user:
        user-name is "x0000",
        user-account is "20402",
        user-passwd is "ritar";


object agent:
        state is "start";
                                         [Attributes of objects       ]
object unixport1;                        [need not be declared.       ]
                                         [they can be created on      ]
                                         [first use                   ]


rule telnet:
IF      the state of the agent is "start"
THEN    send concat("telnet ", nost-name of the remote-system)
                to unixport1
                                         [concat is built-in function ]
        & receive next {"Connections established"} for 15 seconds
                from unixport1 as response of unixport1

                                         [receives from port for      ]
                                         [time limit, quitting sooner ]
                                         [if {pattern} matches        ]
        & set state of agent to "check-telnet";
```

```
rule host:

IF      the state of the agent is "check-telnet"
        & response of unixportl is known
                                      [if no match in rule telnet  ]
                                      [then response is not known  ]
THEN    receive next {login-prompt of remote-system}
                for 60 seconds
                from unixportl as response of unixportl
        & set state of agent to "check remote-host";


rule commands:

IF      the state of the agent is "check remote-host"
        & response of unixportl is known
THEN    send user-name of user to unixportl
        & send user-account of user to unixportl
        & send user-passwd of user to unixportl
        & receive next {prompt of the remote-system}
                for 15 seconds from unixportl
        & return success;             [if this rule fires, then    ]
                                      [RITA monitor returns to     ]
                                      [command level and outputs   ]
                                      [Success! to terminal/file   ]


run;                                  [cycle through rules, testing]
                                      [and firing as appropriate   ]
                                      [see monitor                 ]

  [If all rules are false on any cycle, monitor outputs failure
    and returns to command level ]
```

The record of the interaction resulting from the above command is shown in Example 6 as it would appear on the terminal. Notice that when the user is given the asterisk prompt at the end of the transcript, log-in has been successfully accomplished. This means that all further sends and receives using unixport1 will go directly to and from the remote system. If the attempt to make the connection or to log in to the remote system should fail, RITA would return *Failure* instead of *Success!* just before prompting the user for input.

Example 6

```
There are 70 processes in use (70% loading) and no other
                                                ritas running.

UFE:     14 Jan 77
PARSER:  16 Feb 77
MON:     16 Feb 77

telnet-agent:
[Object remote-system<1> added]
[Object user<1> added]
[Object agent<1> added]
[Object unixport1<1> added]
[Rule telnet added]
[Rule host added]
[Rule commands added]
Success!

*
```

The agent in Example 6 could be expanded by the user to include rules which would be tested and executed only after successful log-in. In order to write such rules, the user should be familiar with the range of responses that the remote system might return. That is, the task that is being automated should be one that is familiar to the user creating the agent. The rules in this example conform to the syntax for rules shown in App. A, which provides an overview of the options available in RITA and serves as a concise graphic index to the concepts and features described in this report.

The simple example given above, showing rules for using telnet to log into a specific host, is not intended to represent a typical agent. For example, rules for handling failure at each juncture where failure is possible have not been included.

Appendix B contains a more detailed example of a useful agent—one that files incoming mail received through an on-line message system. In order to understand the rule set, it is necessary to understand the system with which the agent interacts. Appendix B therefore includes a brief synopsis of the message system's capabilities which are being exercised as well as a flow chart of the logic used to create the agent.

III. RITA CONCEPTS, COMMANDS, AND KEYWORDS

This section presents entries in a dictionary-like format defining RITA constructs and other vocabulary used with RITA. A complete entry has the following format:

entry name (a descriptor in parentheses on the same line indicating whether or not the entry is a part of the RITA syntax, commands, actions, built-in functions, data, values, preinitialized ports, or a UNIX command). The description or definition of the entry.

Form: A canonical form, if appropriate.

Examples: Illustrative examples.

See also: Suggested cross references.

Many entries do not require all of these items; for example, a definition or a cross reference provides sufficient explanation for some of the terms in this manual.

A (syntax). This is one of several optional words which may be used to improve the readability of rules. It is thrown away and/or ignored whenever it is encountered by the system.

See also: article, colon.

abs (built-in function). If the argument evaluates to a string which is a RITA number, *abs* returns the absolute value of the number; otherwise, *abs* gives an error message.

Form: ABS(value)

See also: value, number, arithmetic.

action. A RITA action is any clause in the THEN part of a rule. Actions can effect some change in the data base of objects (CREATE, DELETE, SET, PUT, REMOVE, RECEIVE); interact with an external system (SEND, RECEIVE); invoke the goal-directed monitor (DEDUCE); interact with the user or with a file (SEND, RECEIVE, DISPLAY); set and remove debugging flags (TRACE, UNTRACE, STOP AT, UNSTOP); cause RITA to sleep for a given number of seconds (e.g., DELAY 10 seconds); insert commands, rules, etc., into the command stream (SEND to self); or stop the operation of a pattern-directed monitor (RETURN).

See also: RULE, SET, RETURN, CREATE, DEDUCE, function call, SEND, RECEIVE, DISPLAY, PUT, REMOVE, TRACE, UNTRACE, STOP AT, UNSTOP, DELETE, DELAY.

addition. See arithmetic.

agent. A RITA agent is considered to be a RITA rule set, a set of RITA data objects, and the invocation of a particular monitor to execute them. The agent normally performs some rather circumscribed task, such as getting mail from a remote site, or invokes a number of other agents to perform tasks.

AN (syntax). This is one of several optional words which may be used to improve the readability of rules. It is thrown away and/or ignored whenever it is encountered by the system.

See also: article.

arithmetic. Any string that evaluates to a RITA number (i.e., a sequence of digits optionally including a decimal point, optionally preceded by a plus or minus sign, and with optional leading and trailing blanks) may be used in an arithmetic expression. There are four arithmetic operators, namely + – * /, which represent addition, subtraction, multiplication, and division, respectively. Multiplication and division take precedence over addition and subtraction, as in normal mathematical usage. To override this precedence, the user may group terms with angle brackets <>. Note that expressions may *not* be grouped with parentheses, as these are used only to denote local labels, lists, and argument lists to built-in functions. An arithmetic operator *must* be preceded and followed by a space.

Form:

value + value
value – value
value * value
value / value

Examples:

display 2 + 2;
set the length of the arm to 23 + the diameter of the ball;
display nsubstr(3,2 * <5 + length(response of system)>,
  text of msg);
units of prodn / <<end of oper – start of oper> *
  <basecost of oper-hour + extra-cost of oper-hour>>;

See also: number, isnum, built-in function.

article (syntax). The articles A, AN, and THE may be used to improve the readability of rules. Each is thrown away and/or ignored whenever it is encountered by the system. Articles are stripped from rules upon entry to RITA and inserted by an automatic algorithm when rules are decompiled.

Example:

IF THERE IS A ball WHOSE color IS "green"
  AND THERE IS AN egg WHOSE shape IS "ovoid"
THEN SET THE SHAPE OF THE BALL TO "blue";

is equivalent to

IF THERE IS ball WHOSE color IS "green"
  AND THERE IS egg WHOSE shape IS "ovoid"
THEN SET SHAPE OF BALL TO "blue";

ASCII. See conversion.

assignment. See SET, PUT, object, CREATE.

attribute. A fairly arbitrary string of characters (see *name* for the complete list of legal characters) that serve to define or describe an object. An object can have any number of attributes, but reserved words may not be used for them. An attribute can take on a value (i.e., a string, of which a number is a special case, or a list, see *value)* and can be changed using the SET, PUT, and REMOVE actions. All data in RITA (excluding rules and goals) are stored as object-attribute-value triples. All attributes associated with a particular object must have mutually distinct names. An attribute is defined by setting its value.

Example: Create a conductor whose left-hand is "busy"
& whose right-hand is "free";
Create a ticket-seller whose right-hand is "busy"
& whose left-hand is "free";

See also: name (for restrictions on attribute names), object, value, reserved word, data type.

binding. The object names specified in premises and actions are actually the names of object classes. Before a clause may be evaluated or executed, these object-class tokens must be bound to specific objects of that class by the RITA monitor.

Automatic Binding. Objects are automatically bound by RITA in three different ways: unique binding, existential binding, and deductive binding.

1. Unique binding: If there is only one member of the object class, that member is bound to the object in the rule.

Example: The clause is *the name of the desired-file is "foo.baz"* and there is only one desired-file currently in the system's data base. The class name (desired-file) in the rule is bound to the specific object (desired-file<1>), and all of its attributes and values are carried over and may be referenced throughout the rule.

2. Existential binding: If an existential clause is used (e.g., *there is a file whose location is not known)* and the monitor finds an object that makes the rule true, the first such object found is bound in this clause and in any other clause in the rule (premise or action) that references the same object.

Example: If there is an item whose stock-level is "low" & whose cost is less than 500 & name of the item contains {"copper"} then put the id-number of the item in the orderlist of the company;

Assuming there is only one company in the data base and assuming there are many items, then the first item that satisfies all three clauses in the premise will be bound in the premise and in the action. If no such item exists, the rule will fail to fire.

SPECIAL CASE. If a single rule contains two or more existential clauses that reference the same object class, all subsequent clauses in the premise that reference that object class will be bound to the object in the existential clause of their class most immediately preceding them. In this case, no automatic binding can be done in the action(s) of the rule (unless of course there is only one such object). Therefore, if the object class is referred to in the action(s), it must be bound by the user via the specified form of binding described below under explicit binding.

Example: Rule 1:

```
If: there is a ball(thatball) whose color is not known
  & there is a ball(thisball)
    whose radius is greater than 3
  & the color of the ball (thisball) contains {"gr"}
Then: set the shape of ball(thisball)
    to the shape of ball(thatball);
```

The explicit binding through the use of local labels provides a way to distinguish between the first and second balls found in this example.

3. Deductive binding: A goal rule is selected by the deductive monitor because it has an action that may set an attribute of an object needed either in a DEDUCE clause or in the premise of another goal rule. The object class in that action clause is bound to the same object that is being considered in the calling DEDUCE or premise. In addition, other occurrences of the same object are bound throughout the goal. The object may be unbound and rebound as the deductive search backs up and tries different paths.

Explicit Binding. The user should clarify the desired binding with local labels when automatic binding fails due to ambiguity (see the special case under item 2 above). An example of this type of clarification is given under *local label.* Angle brackets (<>) may be used to refer to a specific object in an immediate action.

Example: Display the name of ball<4>;

This results in the display of the name of the fourth ball created.

EXCEPTION: This form of explicit binding is not available in the immediate action CREATE, but the local label option can be used there.

See also: local label, THERE IS, DEDUCE, deduction, object, attribute, immediate action.

bug. A user bug is an error made by the user in a RITA rule set. A system bug is an error in one of the RITA modules (Front End, Parser, or Monitor) which process the rule set. RITA debugging aids can assist in locating user bugs; these aids include error messages and the ability to trace the evaluation and execution of a rule set at varying levels of detail. If a system bug is encountered or suspected, it should be reported to the RITA system programmer. If possible, such reports should be accompanied by a minimal sequence of com-

mands that cause the problem and a record of the protocol using the UNIX command *proto.* This feedback to the system's implementers would be very useful in making further refinements and improvements to the system.

See also: help, error message, TRACE, SET TRACE, DISPLAY, STOP AT, proto, edit, exit.

built-in function. A number of built-in functions are supplied with RITA to perform arithmetic and string operations on values. Arithmetic operations, or built-in functions that require numerical arguments, are valid only for string values that evaluate to RITA numbers. The function *isnum* is available to the user to determine whether a value is or is not a RITA number. String functions are valid for any string values. The function *islist* allows the user to test whether a particular value is a string or list. The current built-in functions are *abs, clock, concat, eval, floor, index, islist, isnum, lc, length, lindex, max, min, mod, nsubstr, sused,* and *uc.* (Each of these functions is described separately in this section.) Built-in functions are relatively easy to add to RITA. If you need another one urgently, talk to a RITA system programmer. One of the extensions to RITA which is currently under consideration is the addition of a capability that would allow users to define their own functions.

Form: function-name(arg_1, . . .,arg_n)

See also: data type, number, arithmetic.

clock (built-in function). The *clock* function returns the current date and time as a string.

Form: CLOCK()

Example:

```
* display clock();
"Tue Apr 6 18:33:16 1976
"
*
```

Note the newline character at the end of the returned date/time string.

colon. A device (like the articles A, AN, and THE) which may be used freely to increase the readability of RITA rule sets; colons are ignored when read by the system.

See also: article.

command level (of RITA system). When the RITA system is initiated (see *rita* command), the system is in command mode. This is indicated by an asterisk prompt character. In command mode, any RITA command may be issued, including *run, load,* and immediate rules or immediate actions. Note that all RITA commands are terminated by a semicolon (;). A carriage return is used to transmit the command(s) on any line to RITA. The system leaves command mode when the command *run* is encountered in the command stream. An automatic return to command mode will occur under any of the following conditions:

- When all the rules are found to be false.
- When the action of a rule that is fired contains the *return* statement.
- When a STOP AT condition is encountered by the Monitor.
- When a run-time error is encountered (see *error message).*

See also: immediate rule, immediate action, command, continue, run, exit, failure, trace, monitor.

command. Commands are available for use at command level of the RITA system, but they may not be used as actions in rules or immediate rules. They are sometimes referred to as *directives*. The RITA commands are:

run	set unordered
load	set not file (same as nofiles)
exit save	delete (goals, rules, objects)
set ordered	continue
set trace	exit
scratch	verbose
shell	edit
run rule	news
fload	what
quiet	

See also: immediate action, SEND to self, why.

comment (syntax). Comments in RITA rule sets may be included in square brackets anywhere except within identifiers or keywords. Note, however, that the comments are not kept in the internal form of the rules and will not be shown when a rule or object is decompiled.

See also: decompile.

concat (built-in function). Concat evaluates each of the arguments and returns the concatenation of all of them. Non-string arguments result in a run-time error.

Example:

```
* create a foo;
* set color of foo to "blue";
* set hue of foo to "light";
* set shade of foo to concat(hue of foo, " " , color of foo);
* display object foo;
OBJECT foo<1>:
  color IS  "blue",
  hue   IS  "light",
  shade IS  "light blue";
*
```

A *concat* that contains an unknown value will cause an error when the monitor attempts to perform the concatenation.

Form: CONCAT(value,value, . . .,value)

CONTAINS (syntax). The word CONTAINS is used in a pattern-matching premise. The premise "value CONTAINS pattern specification" is true if the scalar value contains a substring that matches the specification (see *pattern*). "value DOES NOT CONTAIN pattern specification" is the negation. The existential forms with THERE IS are also available. (See *THERE IS*).

Form: value CONTAINS pattern specification
value DOES NOT CONTAIN pattern specification
THERE IS [A] [AN] object WHOSE attribute CONTAINS pattern specification
THERE IS [A] [AN] object WHOSE attribute DOES NOT CONTAIN pattern specification

See also: pattern.

CONTINUE (command). The *continue* command is used to restart the monitor. At present, the only difference between this and the *run* command is that when the monitor suspends execution at a rule that has been stopped (via STOP AT), *continue* will continue at that rule (firing it if it is still true), while *run* will restart the scan at the top of the rule list.

See also: RUN, STOP AT, UNSTOP, command level, monitor, return.

control character. RITA responds to three control characters: <CTRL/D>, <CTRL/FS>, and <DEL>. These are defined as follows:

<CTRL/D> (command). Same as *EXIT;* NOTE: The notation <CTRL/x> denotes pushing the x keyboard button while depressing the CTRL shift button.

<CTRL/FS> (command). When all else fails, the <CTRL/FS> quit signal is pretty much guaranteed to cause an immediate crash of the RITA system and all related processes. RITA responds *QUITTING* when it receives the <CTRL/FS> and then waits for about 5 seconds before it dies. After it dies, three large files usually remain in the current directory; these files—called *core*, *core.parser*, and *core.mon*—should normally be deleted.

NOTE: If RITA is not connected to the terminal or if it is connected to it through *proto* (a UNIX program for recording an interactive session with a program such as RITA), then even <CTRL/FS> may not cause RITA to quit and the process may have to be killed from another terminal. See your UNIX systems team.

<DEL> (and <ESC>) (command). The <DEL> interrupt signal is the standard restart for the front end. Unless the parser is running, RITA signals its readiness by immediately reprompting (if the parser is active, the <DEL> will be ignored). Any partially typed command is flushed. Several <DEL>s will force the termination of a load command. A <DEL> while the monitor is running will cause execution to be suspended at the next clean point (between rules); *continue* will continue from such a suspension. A <DEL> will *not* cause termination of an action such as a long display or deduce or receive from the user. As a partial alternative to <DEL>, the user may strike the <ESC> key

followed by a <CR> at any time he is entering a command to cause the partially typed command to be aborted. The front end reprompts when it encounters the <ESC>.

See also: EXIT, help.

conversion (special SEND characters and functions). Two types of conversion are performed in RITA. The first type, called ↑-conversion, is done in order to send ASCII codes (usually non-printing) to remote systems. The second type, called ~-conversion, is done in order to cause RITA to take special port-related actions or to send special signals.

↑-conversions. Communication with many systems often requires the use of odd ASCII characters. RITA provides access to all ASCII codes except NUL (0). When the parser encounters the character "↑ " (up-arrow, ASCII code 0136), it converts the next character into a special ASCII character according to these rules:

1. ↑x (or ↑X) converts to the code CONTROL-X for X some alphabetic character (in either upper or lower case). ASCII codes 1 through 32 (octal) can be obtained this way. Most importantly, a newline (012) is represented as ↑j. (A newline can also be inserted by typing a newline in the string.)
2. ASCII codes 033 through 037 are known as ESC, FS, GS, RS, and US, respectively. These names are printed on the upper-left corner of the appropriate keys of Ann Arbor terminal keyboards. To obtain these codes, the user needs only to use ↑x, where x is the upper-right or lower-middle name printed on that key. For example, one key has the following names printed on it:

   GS }
   ]

   To obtain code 035, also known as GS, the user simply types ↑} or ↑], as either will be converted to GS code. EXCEPTION: The codes RS and US can be obtained only by using the upper-right printable character, since <DEL> and ↑↑ have other meanings.
3. Codes 040 through 0176 are all printable; ↑ must be escaped using ↑↑ and " (inside a string) using ↑.
4. Code 0177 (<DEL>) is obtained using ↑+.
5. The ASCII NUL character cannot be part of a RITA string. However, there is a special escape mechanism, described below.

The ↑-conversions are reversed when the data are decompiled, so that the values may be checked more easily.

~-conversions. These conversions are used to send control functions to the process running on a port, such as killing or interrupting it. The conversions are not done until the string is actually used in a SEND command. If the string

contains any text other than ~-conversion (~ is ASCII code 0176) characters, the converted characters are always sent *first*. This means that the remaining text will probably be going to a closed pipe or a killed process, and such mixing is therefore not recommended. The first two entries below represent characters which are not easily translatable at parse time. All the others are *not* characters at all; they are control codes indicating very special actions.

If the string is being sent to a file, the only conversions that are meaningful are ~~, ~0, and ~$. All other control codes are copied without the corresponding (meaningless) action taking place.

1. ~~ translates to ~. This conversion is used when the tilde is not intended to signify the first of a two-character conversion code.
2. ~0 translates to a NUL (0 byte). This conversion is used because a RITA string cannot itself contain a NUL. NOTE: The inverse conversion is also performed; that is, when a NUL is received, it is translated to ~0.
3. ~$ (no translation). Normally, the SEND action adds a newline to the end of the string being sent. If the code ~$ appears anywhere in the string, however, the code will not be sent, nor will a newline be added at the end. NOTE: If the string consists wholly of ~-conversions, the newline suppression is automatic and no ~$ is needed.
4. ~i, ~q, ~I, ~Q (no translation). These codes cause RITA to send the specified external system an interrupt (<DEL>) or quit (<FS>) signal just as if it had been typed at the terminal.
5. ~e, ~E translates to an EOF. This code causes RITA to send a single end-of-file mark to the specified external system just as if the user had typed <CTRL-D>.
6. ~k, ~K (no translation). This code causes RITA to close the input to the external system and immediately kill it. Next, RITA frees up its table entry for that system. After reading a ~k code, the SEND action aborts; thus, nothing should ever follow this code in a string. After an external system is killed, its name is "forgotten." Any further use of the name in an I/O action will cause a new shell to be created.

See also: SEND, port, NUL.

core dump. See help.

CREATE (action). This action creates a new object in the data base. The object is of type *object*, and all its attributes are considered unknown except for those explicitly initialized. There is no default definition of attributes by object type. Rather, each object in an object class will have an attribute defined for it only when the attribute is set or tested for that particular object or when it is mentioned in a CREATE or in an object description.

Form:
CREATE object
CREATE object WHOSE attribute IS value
CREATE object WHOSE attribute IS value AND WHOSE
  attribute IS value [etc.]

Example:

```
* create a dog;
* display object dog;
OBJECT dog<1>: ;
* if the color of the dog is "red" then set the tag of the dog to
"R";
RULE :
  IF: the color OF dog<1> IS "red"
  THEN: SET the tag OF dog<1> TO "R";
Failure.
* display object dog;
OBJECT dog<1>:
   color IS NOT KNOWN;
```

See also: object description, SET . . . TO (for defining new attributes).

data type. Attributes of objects can take on two kinds of value: string and list. A string is a sequence of characters, and a list is an ordered sequence of values (strings or lists). On input, the list values must be separated by commas and enclosed in parentheses. The RITA number is a special kind of string. On input, character strings must be enclosed within quotation marks; numbers need not be. The maximum magnitude of a RITA number is 10^{30}.

Examples:

```
"this is a string value"
"abc"
("abc",3,"17a")
("another string",("first member of sublist","2n"),"end")
```

See also: value, built-in function, number, arithmetic.

debugging. See bug, TRACE, SET TRACE, DISPLAY, STOP AT.

decompile. When the RITA system interprets statements, it converts them to an internal form. In order to display the stored statements (rules, goal rules, and objects) to the user, RITA must make the reverse transformation. This may cause the displayed items to be slightly different in form from the original input. In particular, comments will be missing, binding of objects will be indicated, and the format may vary.

DEDUCE (action). The DEDUCE action invokes the goal monitor in order to find the value of the specified attribute of a particular object. If the specified attribute is already known, the deduction terminates. Otherwise, the goal rules in the system are scanned for relevant information and applied if possible (see *deduction)*. If no goal rules can help, the monitor will prompt the user for the correct value unless the keyword QUIETLY was specified. The user can then enter the correct value or, if he does not know the value, a carriage return or question mark. In this case, or if the deduction is being done QUIETLY, the value of the attribute remains NOT KNOWN.

NOTE: Whenever a deduction has been completed and an attribute of a particular object has been deduced or found to be NOT KNOWN, any later *deduce* command for the same attribute of the same object will fail to have any

effect unless it is preceded by an explicit command setting the attribute of the object to NOT KNOWN. Failure to have any effect means that RITA will not attempt to perform the deduction.

Form: DEDUCE attribute OF object [QUIETLY].

See also: GOAL, monitor, deduction, App. A (for an example of a deductive rule set), binding.

deduction. The goal-directed monitor may be invoked to deduce the value of any attribute of an object (see *DEDUCE*). The actions of all goal rules are scanned to see if any of them sets the correct attribute of an object. If one does, it binds the goal rule (see *binding*) and attempts to evaluate the premises. If the predicate of the goal rule is true, the relevant action clause *(and only that clause)* is fired, setting the desired attribute of the object. If the rule is false, the monitor looks for other applicable goal rules. If the truth or falsity of the goal rule cannot be determined immediately because various attributes of objects referred to in the premises are unknown, the monitor sets up deductive subgoals to determine those values recursively. If there are no relevant goal rules, the user is asked for the value. He can respond with the value of the desired string, or with a question mark or carriage return if he doesn't know the value (it remains NOT KNOWN), or with the string "why" if he wants to know why the question is being asked (see *WHY*). This questioning of the user by RITA can be suppressed by use of the QUIETLY clause.

If the value of a string is being deduced, the monitor stops after finding the first true goal rule. If the value is a list, it will attempt to fire (only once) all goals that PUT or REMOVE members of the list.

See also: DEDUCE, GOAL, SET, PUT, REMOVE, binding, QUIETLY.

DELAY (action). The delay action evaluates *value* (which must evaluate to a positive integer less than 32768) and puts the program to sleep for that many seconds. It is sometimes useful (but dangerous) for writing timing-dependent system interface rules. The danger arises from the fact that the user can never really know how long to wait, so that a RITA agent could fail if a remote system is slightly slower than usual. The singular form SECOND may be used in place of SECONDS.

Form: DELAY value SECONDS

See also: RECEIVE FOR value SECONDS.

DELETE (action). The *DELETE* action is normally used directly at the command level to remove objects from the data base or to get rid of rules or goals. It may be used as the action of a rule, but this use is considered bad programming style, since it can lead to obscure control structures. When a rule, goal rule, or object is deleted, its space is recycled for other uses.

Form: DELETE ALL RULES
DELETE ALL GOALS
DELETE ALL RULES THAT SET attribute OF object
DELETE ALL RULES THAT TEST attribute OF object

DELETE ALL GOALS THAT SET attribute OF object
DELETE ALL GOALS THAT TEST attribute OF object
DELETE RULE rulename
DELETE GOAL goalname
DELETE ALL OBJECTS
DELETE OBJECT object

DISPLAY (action). The *display* action provides a convenient way to emit information regarding the data base and RITA program to the user's terminal or to a file. The selected rules, goals, and objects are decompiled and printed in such a way that when read back in, the new structures will be identical to what they were. However, the exact wording of the decompiled forms will not necessarily be the same as when the items were first typed in; in particular, all the comments will be stripped.

The user may add TO FILE *file-name* to cause the display output to be sent to a file. If no such file exists, one is created; otherwise, the output is added to the end of the file. To save the partial results of an interrupted RITA session, the user may type the command *display all objects to file save and display all rules to file save and display all goals to file save.*

Form: DISPLAY ALL RULES
DISPLAY ALL RULES THAT SET attribute OF object
DISPLAY ALL RULES THAT TEST attribute OF object
DISPLAY ALL GOALS [THAT . . .]
DISPLAY RULE rule-name
DISPLAY GOAL rule-name
DISPLAY NAMES OF ALL RULES
DISPLAY NAMES OF ALL GOALS
DISPLAY ALL OBJECTS
DISPLAY OBJECT object
DISPLAY value
DISPLAY. . .[TO FILE file-name]

See also: decompile.

division. See arithmetic.

EDIT (command). During a RITA session, a UNIX file is kept containing all the RITA output. This file, called *ufe.output,* contains any syntax error messages incurred by the user's input. The *edit* command is available only when RITA receives input from a terminal (not from a file through redirected input as described in Sec. II). It provides the user with a convenient way to edit and/or save program output and to correct syntax errors in rules. This command calls on an ever-present program which is a version of NED, the new text editor on UNIX.*

After the *edit* command is issued, the editor goes through some machinations and ends with the screen showing the end of file *ufe.output.* This file

*See Walter Bilofsky, *The NED Manual: A Guide to the CRT Text Editor NED,* The Rand Corporation, R-2176-ARPA (to be published).

contains a record of all of the output that has been generated by RITA during the session. It may be scrolled and scanned for syntax error messages or any other information that the user would like to refresh. If the user wishes to create a new file or modify an old file (say, the original input file), the full power of the text editor is available for doing so. An alternate file, *ufe.input*, has also been created. Anything put into the *ufe.input* file will automatically be loaded when the user hits the <DEL> key to return to RITA. Thus, a convenient way to use the *edit* command is as follows: If you load a file and RITA reports syntax errors, issue the *edit* command, scan file *ufe.output* to get the names of the items that had syntax errors, switch (an editor function) to the source file to make corrections, and pick and put the corrected rules from the source file into file *ufe.input* so they (and only they) will be reloaded. NOTE: The files used by the *edit* command are always deleted by RITA when exiting (*ufe.input* is deleted immediately after being loaded).

Form: EDIT;

See also: nofiles, LOAD.

EQUAL TO (syntax). See predicate.

error message. Error messages can be generated by the user front end (UFE), the parser, or the monitor (at run time).

UFE errors. The UNIX operating system has upper limits on the number of files that can be open for individual users and for all users on the system. The UFE will give an appropriate error message and exit if there are no file descriptors available when it tries to create communication pipes to the parser and monitor. Similarly, if the UNIX limit on processes is reached so that the RITA processes cannot be created, the UFE will exit with the error message *can't fork.* A number of other error conditions (e.g., monitor or parser programs unavailable, or unable to create temporary files) are reported with appropriate messages.

Parser errors. The parser gives appropriate error messages when invalid rules, objects, or commands are input. Normally, the location of the error is indicated by echoing the input and putting the string [***] at the offending location. A common parsing error is caused by the use of a RITA reserved word (see *reserved word)* as the name of a rule, object, or attribute.

Monitor errors. The monitor recognizes a number of error conditions, many of which should never occur. The most common error is attempting to use a rule whose objects are not defined (resulting in the error message UNBOUND). A list of illustrative user-caused error messages follows:

```
NEEDSCALAR: A list was used where a scalar was expected.

NOTKNOWN:  Inappropriate use of unknown value.

NEEDINT: A non-numeric string was used where a number was expected.

STRLIST: List in RECEIVE NEXT or pattern may not have sublists.
```

```
DIVZERO: Zero used as second argument of a division or mod function.
ZSTRING: Strings are (now) 1-indexed ONLY.  You used a Ø-index.
BADARGS: A built-in function was called with wrong number of arguments.
LARGENO: You specified a number whose magnitude is too large.
BADZERO: A zero was used where a positive number was expected.
NEGNUM: Inappropriate use of negative number.
UNIMPLEMENTED: a particular RITA feature is not yet implemented.
            See a systems programmer if you need it soon.
UNBOUND: There is no way to establish a binding for some object.
         (see binding)
OVERBOUND: Trying to create an object previously bound (see binding)
PORTS: Only three external systems may be controlled
       simultaneously (see port).
SCALSS: Trying to do a list selection on something that is not a list
TOOBIGVAL: Trying to access a list member that does not exist.
UNKVAL: Unknown value in a function; must be able to evaluate it
NOMEMB: There IS no such member of that list
MEMBF: IF value IS IN list requires a scalar and a list
SMLABEL: You tried to enter a specific object, and we already
         know about more of them.
NORES: Out of system resources: UNIX has run out of open files,
       pipes, or some other commodity
BADDEL: rules and goals may be deleted only at command level
SYMTAB: Too many symbols used (the maximum number allowed is 1024)
BADFILE: Cannot send to the specified file
```

If the RITA system is greatly confused, it will give the error message *system bug* and will print some information that will be useful to the RITA system programmer.

See also: bug.

eval (built-in function). The two argument values are evaluated. If the second argument evaluates to a string that is a legal object name, and if there is only one object of that class, and if the first argument evaluates to an attribute of that object which has been set and which is not a list, then *eval* returns the value of the attribute of that object. If the object exists, is unique, and the

attribute name is a valid one but is *not known, eval* returns NOT KNOWN. If the second argument evaluates to a legal object which is not unique, then *eval* returns NOT KNOWN. If the first argument is a list, *eval* returns an error.

Form: EVAL(value,value)

See also: name, value.

EXIT. One of the most important things to know about the RITA system is how to get out of it! The *exit* command is the usual way. Typing an end-of-file (<CTRL-D> from the terminal) to the front end will also work.

When the front end is about to exit, it says "exiting." NOTE: the front end performs a lot of cleanup functions when exiting; if RITA should ever exit without the exiting message, something is wrong and the user should contact a RITA system programmer.

After normal exiting, no additional files should be present in the user's directory. When running, RITA produces a *ufe.output* file for editing and a *rita.history* file for the explanation systems. It is occasionally useful to preserve these files after a RITA session, and this may be done by issuing the *exit save;* command. Also, RITA sometimes encounters system errors that cause it to die (generally with a message such as *Monitor death* or *Symbol table overflow)*; when this happens, the user is prompted with a *Save files?*, to which he should respond "y<cr>" to preserve or "n<cr>" to delete these files.

Form: EXIT [SAVE];

explanation subsystem. See WHY, DEDUCE, WHAT.

external system. One of RITA's strongest points is its ability to control external systems. Ports are available for communicating simultaneously with up to three such processes. A port is initialized by executing a SEND or RECEIVE action specifying that port name. A new port is connected to a standard UNIX shell (see SHELL (I) in the UNIX Programmer's Manual), and the value specified in the SEND is written directly to the shell. The value should be a UNIX command (typically Telnet, Lisp, or another RITA). Combinations of RECEIVEs and SENDs are used to control and follow the progress of the external process.

See also: SEND, RECEIVE, SELF.

failure. See monitor (rule-directed), RETURN (Success/Failure).

"FALSE" (value). Some built-in functions such as *isnum* and *islist* return the value "FALSE" if the condition being tested is not satisfied. Note that "FALSE" is a literal string and is in upper-case letters.

See also: built-in function, "TRUE".

FLOAD (command). The *fload* command is similar to the *load* command but runs a bit faster. The speed increase is achieved by bypassing all of the front-end error-handling/buffering code. Thus, files to be *floade*d must already be

known to be error-free (i.e., they must be loadable). Furthermore, the file may contain *only* rules, goals, objects, and the command(s) *set (un)ordered.* All other commands will be ignored (e.g., *trace, run)* or will cause errors (e.g., *load, fload, scratch).* Any errors in an attempt to *fload* a file will cause immediate suspension of the attempt, and an *error in file* message will be displayed (but no explanation). Because of this, *fload* should be used only for already debugged rule sets.

A file may not be *fload*ed by mentioning it as an argument after the *rita* command; these files are always loaded. However, a load file may be created that simply contains a *fload* command.

Fload is not significantly faster than *load* in the current version; it may eventually be dropped, so it is unwise to rely on it.

Form: FLOAD file-name

See also: LOAD.

floor (built-in function). If the argument evaluates to a string which is a RITA number, *floor* returns the largest integer less than or equal to the number. Otherwise, *floor* gives an error message.

Form: FLOOR(value)

See also: value, number, arithmetic.

front end. See rita.

function call (action). An action or a premise may include a function call. For example, a premise might use the returned value of the function call as the value to be tested in a predicate, as in

length(name of person) is greater than 12

In an action, function calls can be meaningfully combined with the SET . . . TO and DISPLAY actions, as in

set name of officer to concat(rank of officer,lastname of officer)

Value(s) returned by function calls are described in detail under the entry for each built-in function. If a function call results in an error, execution is suspended and RITA returns to command level. See *built-in function* for a complete list of these functions.

Form: function(arguments)

See also: built-in function.

GOAL (command). The GOAL command enters a rule into RITA that will be accessed only by the goal-directed monitor during a deduction. Almost all premises available to a RULE are allowed (see exception below), but only SET and PUT actions are allowed in a GOAL. When a GOAL is executed, only the one relevant action clause through which it was chained into the current deduction is fired; i.e., there are no side effects in the actions of GOALs.

Because the object class for which an attribute is being deduced is bound to a particular object at the time of the DEDUCE, a THERE IS clause for the object class being deduced should not be used in the premise of a GOAL that might be used in that deduction, as this will cause automatic binding which conflicts with previous binding.

For example, the following rule and goal pair would cause such a conflict:

> Rule one: IF there is a dog whose pedigree is not known
> THEN deduce the stable of the dog;
> goal A: IF there is a dog whose breed is "Dalmatian"
> THEN set the stable of the dog to "Sunset Kennels";

In this example, the first dog found whose pedigree is not known is bound in the deduction. Goal A is then perceived to be a goal which might set the stable of the dog. In testing the premise of the goal, the first dog whose breed is Dalmatian is found and bound. Since this may not be the same dog that was bound in the deduction originally, there is a conflict which, if allowed, could lead to inappropriate results. Therefore, if such a situation is encountered during a deduction, an error condition will occur.

Form: GOAL goalname IF premises THEN actions;

See also: name, premise, action, DEDUCE, deduction, RULE, SET, SET, PUT.

GREATER THAN (syntax). See predicate.

help (what to do in various emergencies). Procedures are available in RITA for dealing with such user problems as the following:

Problem: Everything is messed up or hung and you can't get out of RITA.
Solution: Hit <CTRL/FS>. It will almost always get you out. (See the entry *control character* for exceptions and further help.)

Problem: You just typed in a long rule with a syntax error in it and you don't want to type it in again.
Solution: Use the EDIT command; it will display the echoed rule with the syntax error in it so that you may correct it.
Problem: Your program is looping and you want to stop it without exiting from RITA.
Solution: Hit the DEL key. RITA will stop after the next attempted rule. (It sometimes needs more than one DEL.)

Problem: RITA exited strangely and gave core dumps.
Solution: Try to isolate the problem, then contact a system programmer, preferably with a record of the interaction (use UNIX program *proto* to create one).

Problem: RITA gave an error message that it admits is a system bug.
Solution: Save the input files and output files and give them all to a system programmer.

Problem: RITA is asking a question that you did not put in, and you don't know why RITA would want to know the answer.

Solution: It's doing a deduction (see *deduce, deduction)* and needs that question answered to satisfy a subgoal. Answer it with the string "why" to find out which subgoal it is trying to satisfy (see WHY).

See also: what, why, error message.

IF (syntax). The keyword IF is used to introduce the premises of a rule or goal.

See also: rule, goal, premise, immediate rule.

immediate action (command). Any legal RITA action may be given at the RITA command level for immediate execution. Such immediate actions are not contingent on the premise in a rule. They include the delete action and those listed under the entry *tracing and debugging.* Actions are legal commands even when they are taken from an input file (redirected input, see Sec. II) or from a LOAD file.

Example: A user wishes to preinitialize I/O channels by the use of immediate actions in a separate LOAD file to facilitate switching between, say, a simulated port and the real thing; this LOAD file could consist of

```
send "net" to net;
send "cat nyt" to nyt [simulate nyt];
send "initialized" to user;
```

Multiple actions may be connected by the word *and,* but they do not need to be, as they are executed one after another anyway. Immediate actions must end with a semicolon, just as commands do.

When actions are used as commands—and *only* in this case—the user is allowed to specify bindings on objects. Bindings are specified by an integer surrounded with angle brackets (<>) following the object name (and local label, if any); for example, *delete ball<3> and set the color of box<2> to concat("very",color of box<1>);*. The way to determine the binding number of a specific object is either by decompiling a rule that has its bindings intact (see *binding),* or by displaying all objects, or, better, displaying object objname; ball<n> refers to the nth ball to have been created in the data base. NOTE: If bindings are encountered anywhere but in an immediate action, they are simply ignored and no error messages are produced.

See also: SET, RETURN, CREATE, DEDUCE, function call, binding, SEND, RECEIVE, DISPLAY, PUT, REMOVE, TRACE, UNTRACE, STOP AT, UNSTOP, DELETE.

immediate rule (command). Occasionally, a user either does not know a binding or would like to know if some predicate is true of the data base. In each case, he would use an immediate rule. An immediate rule has no name and no "rule" header; it starts with the keyword *if.* The entire rule syntax is available. It is especially important to note that either the left-hand side of a rule or the right-hand side or both may be left empty. Thus, if the user wants to

test premises but not actually do anything, he can say "if p1 and p2 . . . and pn then;". An immediate rule with a null left-hand side is like an immediate action, except that no bindings may be specified.

An immediate rule is immediately evaluated, and if it is true, its actions are executed. In any event, the rule is then decompiled with bindings displayed and either *Success* or *Failure* is printed to indicate whether or not the rule fired (was true). For example, the decompiled immediate rule 'rule : if there is a ball<2> whose color is "red" and whose size is "small" then;' indicates that such a ball does indeed exist and that it is the second ball in the data base.

See also: premise, predicate, binding.

index (built-in function). Both values should evaluate to strings. If the first string is a substring of the second, the function returns the location of the first match. The string is one-indexed, i.e., it starts with character position 1. Index returns the string "FALSE" if there is no match.

Form: INDEX(value,value)

Examples:

```
* display index("baz","foobaz");
4
* object ball color is "blue-green";
[Object ball<1> added]
* display index("gr",color of ball);
6
*
```

See also: lindex, nsubstr, length, pattern, CONTAINS, RECEIVE.

integer. An integer is a whole number, that is, a number without a fractional part. Some features of RITA, such as some of the built-in functions, require integers. A RITA integer is defined to be a RITA number not containing a decimal point, with absolute value not greater than 32767. If a non-integer number is supplied where RITA requires an integer, it will be rounded to the nearest integer.

See also: number, arithmetic, built-in function.

IS [NOT] IN (syntax). See predicate.

islist (built-in function). The argument is evaluated. If it is a list, the function returns the string "TRUE"; otherwise, it returns "FALSE".

Form: ISLIST(value)

Examples:

```
* display islist(3);
"FALSE"
* display islist("foo");
"FALSE"
* display islist(("foo"));
"TRUE"
* display islist(("foo","baz"));
"TRUE"
```

```
* display islist(( ));    [empty list]
"TRUE"
*
```

See also: lindex.

isnum (built-in function). This function returns "TRUE" if the value is a RITA number; otherwise, it returns "FALSE".

Form: ISNUM(value)

See also: number, data type.

I/O. See RECEIVE, SEND, DISPLAY, external system, port.

keyword. A keyword is a word having special meaning to RITA. No special care need be exercised by the user with regard to keywords. However, a subset of these words are reserved and may not be used as names of objects, attributes, rules, or goals. These are listed under *reserved word.*

KNOWN (syntax). See predicate.

label. See local label.

lc (lower case) (built-in function). The argument should evaluate to a string. If so, the returned value is the same string with all upper-case letters shifted to lower case. If not, a run-time error message is given.

Form: LC(value)

Left-hand side. This term refers to the left-hand-side monitor, which is either of the pattern-matching monitors also known as rule-directed monitors, ordered and unordered. (The right-hand-side monitor is known as the goal-directed monitor.) *Left-hand side* may also refer to the side of a rule that consists of the rule's premises. (This terminology is derived from language used to describe grammar rules, which are related to production systems.)

See also: monitor.

length (built-in function). The argument should evaluate to a string. If so, the returned value is the number of characters in the string.

Form: LENGTH(value)

LESS THAN (syntax). See predicate.

LHS. See left-hand side.

lindex (built-in function). The function *lindex* finds the location of a member of a list. The second argument should evaluate to a list; the topmost level of this list is searched for the first occurrence of the evaluated first argument, and the (1-indexed) position of the member is returned. If the first argument is not a member of the list, the string "FALSE" is returned.

Form: LINDEX(value,value)

Examples:
```
* display lindex("foo",("string1","baz","foo","arf"));
3
* display lindex((1,2),(3,4,(5,(6,7)),8,(1,2),9));
5
* display lindex((6,7),(3,4,(5,(6,7)),8,(1,2),9));
"FALSE"
*
```

See also: index.

list. See value, PUT, REMOVE, islist, size, lindex.

literal. A literal is a quoted character string or a number. Rather than allowing a value that will evaluate to a string or number, RITA syntax sometimes requires use of the literal itself. A literal list is a list of literals.

LOAD (command). The *load* command is followed by the name of a UNIX file and causes the RITA front end to treat the contents of the file exactly as if it were just being typed in on the terminal. The file is read in until its end-of-file (it continues reading the file even after syntax errors).

All the commands described in this reference manual may also occur in a file to be loaded. In particular, a *run* command may be issued within the file being loaded, so that the user need not issue it directly. However, there is one restriction: *load* commands can be nested four times, but no more; that is, within reasonable limits files can contain commands that load other files. Nothing can appear on the same line after a *load* command when the command is typed in from a terminal or in a send-to-self (normally, more than one command can be typed on a line).

Load commands can be nested, which provides a great convenience, as the files loaded may contain any legal commands. For example, a typical file to be loaded may consist of

```
quiet;
load foo.rules;
load foo.objects;
trace all rules;
verbose;
send "Ready" to the user;
```

Files may be loaded automatically at the start of a RITA session by listing their names as arguments after the RITA command.

At present, text cannot be put in a load file that is intended to answer questions asked by a program after a *run* command is issued (i.e., the user must still type in the responses asked for). This includes answering the replacement confirmation message.

Form: LOAD file-name

See also: FLOAD, rita.

local label (syntax). In a rule or goal, the name of an object may be followed by another name in parentheses. This name is a local label which may be used within this rule to identify multiple references to the same object. It is particularly useful for referring to an object found by a THERE IS clause.

Form: attribute OF object (string)

Example:

```
* display rule 1;
RULE 1:
  IF: THERE IS A ball(thisball) WHOSE radius IS greater
THAN 3
    & THE color OF THE ball (thisball) CONTAINS {"gr"}
    & THERE IS A ball(thatball) WHOSE color IS NOT
KNOWN
  THEN: SET THE shape OF ball(thisball)
    TO THE shape OF ball(thatball);
*
```

The local label provides a way to distinguish between the first and second balls found in this example.

See also: binding.

lower case. See lc.

max (built-in function). If all the arguments evaluate to RITA numbers, *max* returns the maximum value.

Form: MAX($value_1$,. . .,$value_n$)

min (built-in function). If all the arguments evaluate to RITA numbers, *min* returns the minimum value.

Form: MIN($value_1$,...,$value_n$)

mod (built-in function). If the arguments evaluate to valid RITA numbers and value2 is not zero, *mod* returns the remainder of value1 divided by value2.

Form: MOD(value1,value2)

monitor. Three monitors of two types are currently implemented in RITA:

* *deductive (or goal-directed).* The deductive monitor is invoked by an explicit DEDUCE action in a rule executed by one of the other monitors. (See *DEDUCE* and *deduction* for a complete description of its operation.) It is also called the goal-directed or right-hand-side (RHS) monitor.

* *pattern-matching (rule-directed).* There are currently two rule-directed, or left-hand-side, monitors (ordered and cyclic). The cyclic monitor is the default monitor when the *run* command is given in the absence of a previous *set ordered* command. When either of these monitors is invoked, it will test rule premises and execute the actions of true rules until it makes one complete pass through all the rules and finds them all false. At this time it will print

"Failure" and return to command mode. (See *STOP AT* and *RETURN* for additional ways to return to command mode.)

In the ordered monitor, the rules are considered to be ordered. This monitor starts at the top of the list of rules, testing the IF part of each. When it finds a true rule, it executes all the actions of that rule and begins again at the top of the rule set. This process continues until either an action terminates it (RETURN), a user-defined break occurs (STOP AT), or no rule is true. At that point, control is returned to the user. To use this monitor, the *set ordered* command must be issued before a *run* command.

The cyclic, or unordered, monitor also considers its rules to be ordered, despite its name. Its operation is the same as that of the ordered monitor, except that when a true rule is found, the next rule to be considered is the next one on the list, rather than the first rule on the list. The ordered and cyclic monitors are also called pattern-action monitors, forward-driven monitors, or left-hand-side (LHS) monitors. The cyclic monitor is the default monitor, and it will be invoked on a *run* command unless the *set ordered* command has previously been given. To change back to the cyclic monitor, the *set unordered* command may be used.

Many other monitor strategies could be implemented. We do not rule out any of them in the future development of RITA, but we have found these three to be adequate for most purposes.

The deductive monitor is used when a particular datum is needed. It is most useful in a static situation, where the data, once deduced, will keep the same values. The LHS monitors are particularly useful for dynamic situations such as interactions with external systems, where the program receives data and takes actions depending on those changing data. The cyclic monitor has the advantage that its rules are more independent than those of the ordered monitor, but these independent rules normally require more premises.

See also: SET ORDERED, SET UNORDERED, deduction, DEDUCE, rita, GOAL, RULE, explanation subsystem.

multiplication. See arithmetic.

name. There are two kinds of names in RITA. The first kind, which includes rule names, goal names, file names, port names, pattern labels, and local labels, may be either quoted strings or unquoted strings. If they are quoted strings, they may contain any arbitrary characters, including blanks. If they are unquoted strings, they may not include a blank or any of the following characters: < > [ ] { } ( ) & : ' ; , " ?

Examples: 1
"select fastest"
printfile
3A

The second type, which includes function names, object names, and attribute names, must be unquoted non-numerical character strings not containing blanks and not containing any of the following characters: < > [ ] { } ( ) & : ' ; , " ?

Examples: ship
arrival-time
A3

See also: pattern labels (for special use of single quotes).

NEWS (command). The *news* command displays the RITA news file using the UNIX program *la*. A short list of instructions for running *la* appears at the beginning of the news file. This file lists in inverse chronological order all the changes and additions to RITA that affect the user.

Form: NEWS;

NEXT (syntax). See RECEIVE.

NOFILES (command). As mentioned under the *exit* command, RITA normally produces two files (ufe.output and rita.history) while running. These files tend to grow large over time. To avoid paying the cost of writing these files, the user may issue the *nofiles* command, which deletes these files and prevents further output to them. After a *nofiles* command, the *edit* and *what* commands are also disabled, since they need the deleted files to operate. Also, any *Save files?* prompts are automatically answered in the negative, since there are no files to save. The commands *nofiles* and *set not file* are equivalent.

Form: NOFILES
SET NOT FILE

See also: EXIT.

NOT. See predicate.

NOT IN. See predicate.

NOT KNOWN. See predicate.

nsubstr (built-in function). This function extracts a specified substring from a target string. The first argument should evaluate to a positive integer, and the second argument should evaluate to a non-negative integer. If either of them evaluates to a number other than an integer, it is rounded to the nearest integer. The first argument indicates the starting location of the substring (1-indexed) in the target string; the second argument gives the length of the substring; the third argument evaluates to the target string. If (arg1 + arg2 − 1) exceeds the length of the string, then the returned value is the rightmost portion of the target string, starting at the arg1th character. If arg1 exceeds the length of the string, then the returned value is the null or empty string ("").

Form: NSUBSTR(value,value,value)

Examples: * create a ball whose color is "blue-green";
* create a person whose height is 5;
* display nsubstr(6,height of person,color of ball);

```
"green"
* display nsubstr(4,3,"foobaz");
"baz"
* display nsubstr(4,3,"foob");
"b"
*
```

See also: index, pattern, CONTAINS, receive.

NUL. A NUL is a zero byte. A RITA string is not allowed to contain a NUL, because RITA uses a NUL to mark the end of a string. (However, the user need not be concerned with this fact.) A NUL can be sent in a SEND action by representing it as "~0". Similarly, when a NUL is received in a RECEIVE action, it is converted to "~0" and treated as two characters.

See also: conversion, RECEIVE.

number. A RITA number is a string consisting of one or more digits, optionally including a decimal point, optionally preceded by a plus or minus sign, and having optional leading and trailing blanks. Its magnitude must be less than 10^{30}. Note the problem of round-off error and loss of significance: An arithmetic operation involving numbers with many digits may not yield an exactly correct answer, but the answer will always be correct to within a very small percentage.

See also: data type, integer, arithmetic, value.

object (data). All data in RITA are stored in objects. The data base consists of a collection of object-attribute-value triples. Each object has a type (i.e., a class name) and an arbitrary number of attributes. The object type is a fairly arbitrary string of characters (e.g., ball, external-system) but cannot be a reserved word. Objects are entered into RITA using the CREATE or object description syntax and are deleted with the DELETE command. There may be several (or many) objects with the same type, and they may be accessed associatively by testing their values in a THERE IS clause:

> Rule 1: If there is an external-system (S) whose operating-system is "TENEX", then set the name of the current-system to the name of external-system (S);

Objects of the same type need not all have the same attributes; and the order of attributes (transparent to RITA) need not be the same in objects of the same type.

There are no predefined objects in RITA.

See also: name (for limitation on object names), attribute, CREATE, object description, DELETE, THERE IS, value (for restrictions on OBJECT syntax).

object description (command). Objects may be entered into RITA by typing them in, using the object description syntax. The new objects are always added to

any others—replacement does not occur. An equivalent method of entering objects is to use the CREATE syntax, which is more readable. For example,

```
object person id is "37",
    name is "H. Q. Bovick",
    telephone is "810-3876",
    location is "home";
```

could also be created in the following way:

```
create a person whose id is "37"
    and whose name is "H. Q. Bovick"
    and whose telephone is "810-3876"
    and whose location is "home";
```

Form: OBJECT objectname
OBJECT objectname attribute IS literal
OBJECT objectname attribute IS literal,
attribute IS literal,
attribute IS literal
OBJECT objectname attribute IS (literal-$_1$,...literal-$_n$)

See also: CREATE, SET . . . TO, name, value (for restrictions on OBJECT syntax).

OR (syntax). See premise.

parser. See rita.

pattern label. A pattern label may follow any pattern in a pattern specification. It is used to store, for temporary use, the string that matches the pattern. A pattern label is set only after a successful match of the pattern specification and can then be accessed only within the clauses in the rule subsequent to its setting. A pattern label is considered not known in any other circumstance.

Pattern labels must be enclosed in single quotes. (See *name.*)

See also: pattern.

pattern. A pattern specification may be used in a premise with the phrases CONTAINS or DOES NOT CONTAIN to test for the existence of that pattern specification in a (scalar) value, known as the target string. A pattern specification may also be used in a RECEIVE action. The following discussion applies specifically to "value CONTAINS pattern", though RECEIVE behaves in a similar manner.

A RITA pattern specification describes the characteristics of a string. It is composed of a sequence of one or more patterns (described below). RITA attempts to match each successive pattern to some substring of the target string. A pattern specification may be general enough to describe a large class of strings or it may be so specific as to describe a unique string.

A pattern specification is always enclosed in curly braces. It may begin with the phrase "START FOLLOWED BY" and may end with the phrase "FOLLOWED BY END". The meaning of these phrases is explained below.

Patterns in a pattern specification are separated by the phrase "FOLLOWED BY". Each pattern may be followed by a pattern label in single quotes. The pattern label is used to store temporarily the substring that matched the preceding pattern. For example,

> IF the name of the employee contains {ANYTHING 'firstname' FOLLOWED BY "Smith"}
>
> THEN put 'firstname' into given-names of smiths;

In this example, if the name of the employee is "John Smith," then 'firstname' is set to "John" upon successful match of the pattern specification to the value. Pattern labels have the value "NOT KNOWN" outside the rule in which they are set, and they are known within that rule only in premises or actions that follow the successful match in which they are set.

A pattern specification contains a target string if each pattern matches some substring of the target string, the substrings occur in the same order as the patterns, and there are no gaps between the substrings. The first pattern in a pattern specification need not match an initial substring of the target string. Similarly, the last pattern need not match a terminal substring. For example, the pattern specification

> {ANYTHING FOLLOWED BY "Movement Reports" FOLLOWED BY ANYTHING}

is equivalent to the pattern specification

> {"Movement Reports"}

Some patterns may be matched by strings of different lengths. For example, in the pattern specification

> {"Date" FOLLOWED BY ANYTHING FOLLOWED BY "1976"}

the substring that matches ANYTHING may be of any length (including 0) and all of the following strings contain this pattern:

> "Date: August 11, 1976"
> "... On the Date of May 3rd 1976, the first"
> "Date 1976"

RITA will always find a match if there is one. In cases where more than one match is possible, RITA will report the first one that it finds. To predict which match will be found in ambiguous cases, it is necessary to understand the order in which RITA proceeds. Variable-length patterns will initially be matched either to the longest or the shortest possible substring (shortest for ANYTHING, longest for SOME or ANY), and if subsequent patterns fail to match, RITA will retry by either shortening or lengthening this substring, continuing if necessary until all possibilities have been examined. For example, if the string "My Dear Henrietta," is compared with the pattern specification

> {"Dear" FOLLOWED BY ANYTHING FOLLOWED BY ","}

the pattern matching proceeds as follows. The first pattern in the pattern specification ("Dear") is tested against the first position in the target string, and the match fails. The search position is advanced to the second position in the target string, and the match is retried. Upon the fourth try (starting at the fourth character in the target string), the match succeeds ("Dear" in the pattern matches "Dear" in the target string). Now the second pattern (ANYTHING) is tried, starting at position 8. This succeeds immediately, matching the null substring. The search position is still 8. Next the third pattern (",") is tried. It fails, since there is no comma in position 8. RITA then retries by going back to its last specified variable-length pattern (ANYTHING) and lengthening the corresponding substring, matching the "ANYTHING" to the comma in position 8. The search position is advanced to position 9 and the third pattern is tried again, etc. This continues until the substring matching ANYTHING is long enough so that the third pattern matches the comma in the target string, terminating the pattern search successfully. Had there been no comma in the target string, RITA would have kept lengthening the substring corresponding to "ANYTHING" until the target string was exhausted. This would have concluded the search unsuccessfully, since in this example there are no other variable-length patterns to lengthen or shorten. The search procedure is completely exhaustive: When a pattern fails to match the target string at the current position and there are one or more variable-length patterns before it, RITA will always retry until all possibilities have been examined. For example

{SOME IN "0123456789" 'prefix' FOLLOWED BY "395"}

would be successfully matched by either of the following strings, and 'prefix' would be set as shown after the match:

"3923950" 'prefix' = 392
"0395" 'prefix' = 0

The types of patterns are listed below.

value: Value may evaluate to any legal RITA value except a nested list; i.e., a RITA number, a quoted string, or a (1-level) list of RITA scalars (RITA numbers and/or quoted strings). When a pattern is a scalar value, it is matched if the next substring in the target string is the scalar value. When the pattern is a list of values, it is treated as a list of acceptable alternative scalar values, any one of which would be considered a match of the pattern. For example,

"ABC" contains {"AB"}
"A" contains {("B","A")}

integer CHARS, integer LINES: This pattern is matched by the occurrence of exactly integer contiguous occurrences of any characters (including non-printing characters such as newline or "↑j", see *conversions)* or the occurrence of exactly integer sequential lines. If the substring being tested against the pattern has fewer than integer characters (or lines), the match fails. Note that blank, newline, and other non-printing characters are considered to match in

the integer CHARS specification, and blank lines are considered to match in the integer LINES specification. Integer must evaluate to a non-negative RITA integer. (If it evaluates to a non-integer number, it will be rounded to the nearest integer. This is also true of the other patterns that call for an integer.) "Line" or "lines" and "char" or "chars" when not preceded by an integer have the default value of 1 line or 1 character, respectively.

ANYTHING: ANYTHING is matched by the null string (empty string) and/or any and all characters up to the substring matching the next pattern specified. For example,

{"Employee name:" FOLLOWED BY ANYTHING FOLLOWED BY "Social Security Number:"}

would be matched by any of the following strings:

"Employee name: Tom Jones   Social Security Number:"
"Employee name:Social Security Number:"
"Employee name: Marvin Gardens   Age: 102   Social Security Number:"

SOME [NOT] IN string: This pattern is matched by at least one and possibly more contiguous occurrences of any character [not] in a string. Note that the occurrences need not be in the same order as they appear in the string; e.g.,

"2001" contains {SOME IN "0123456789"}

and

"abcde" contains {SOME NOT IN "0123456789"}

The string in this pattern need not be a literal string but must evaluate to a string.

ANY [NOT] IN string: This pattern description is exactly the same as "SOME [NOT] IN string" except that it is considered to be matched by zero or more contiguous occurrences of the characters [not] in string.

integer CHARS [NOT] IN string: Integer must evaluate to a non-negative RITA integer; string evaluates to a string. This pattern is matched by an occurrence of exactly integer CHARS [not] in the string. For example, if digits of system is "0123456789",

5 CHARS NOT IN digits of system

is matched by the next five consecutive characters if they are non-numeric. Note that the match fails if there are less than five characters left in the substring of the target string. When this pattern is specified without the integer preceding it, the default is 1 char [not] in string, regardless of whether the keyword *char* or *chars* is used.

POSITION integer: Integer must evaluate to a non-negative RITA integer. This pattern is used to specify the 1-indexed position relative to the start of the target string. This pattern will succeed only if the pattern search is at the

position specified by the integer. (For the meaning of search position, see the general description above.)

For example, the pattern specification

{"A" FOLLOWED BY POSITION 7}

will be matched by any string that contains an "A" in position 7. The pattern specification

{POSITION 7 FOLLOWED BY "B"}

will be matched by any string that contains a "B" in position 8. The pattern specification

{SOME IN DIGITS OF SYSTEM FOLLOWED BY POSITION 7
FOLLOWED BY 1 CHAR NOT IN DIGITS OF SYSTEM}

would be matched by many strings, for example,

"0001234A"
"0000001B12"
"   35 (thousands)"

Note that a pattern specification that contains a *POSITION integer* pattern can be successfully matched only by a string whose length is at least *integer*. "Position 0" is a synonym for "START".

Form: {[START FOLLOWED BY] pattern [pattern-label] FOLLOWED BY pattern [pattern-label]. . . [FOLLOWED BY END]}

Examples: 1. The following pattern specification finds the first integer in a string and stores it in a pattern label called 'first-number':

{SOME IN "0123456789" 'first-number'}

To find the last integer in a string, we write

{SOME IN "0123456789" 'num' FOLLOWED BY ANY NOT
IN "0123456789" FOLLOWED BY END}

2. To retrieve the first floating-point number (that is, a number with a decimal point, such as 3.14) between the 21st character and the 40th character of a target string, we use the pattern specification

{POSITION 20 FOLLOWED BY ANYTHING FOLLOWED BY
SOME IN ".0123456789" 'num' FOLLOWED BY ANYTHING
FOLLOWED BY POSITION 40}

3. To find the first word in a text that ends with ed, ly, or ing, set alphabet of system to "AaBbCc. . .Zz" and use

{SOME IN alphabet of system 'root' FOLLOWED BY ("ed", "ly", "ing") 'suffix' FOLLOWED BY SOME IN " . , ; : ( ) ↑j " }

where the last pattern includes whatever punctuation characters are to be allowed. To capture the first word thus found, set some attribute to concat ('root', 'suffix').

See also: conversions (for meaning of "↑j"), receive, integer, list.

Pausing.... This is a message that is emitted by RITA during the output of TRACE data to the terminal. It helps the user to control the rapid flow of trace output to the terminal, as he must enter a carriage return before another screen-size chunk of trace information will be printed to the screen.

pipe. See external system, port.

port. RITA's interactions with external systems take place on communication paths called ports. A port is set up and given a name by using that name in a SEND or RECEIVE action, e.g., *send "telnet" to unix-port* would set up a port named *unix-port*. The first use of a port generates a UNIX shell process; if the action is a SEND, the value is sent to the new shell. In the example above, the UNIX subsystem *telnet* would be started in the shell connected to *unix-port*. Subsequent SENDs and RECEIVEs using that same port name will talk and listen to the same shell.

A maximum of three ports (not including the user and self ports) may be active at the same time. This means that RITA may be talking simultaneously to three distinct external systems (e.g., a data base system on the ARPAnet, the New York Times Information Bank, and a desk calculator system under UNIX). When an interaction with one system is completed, the port may be released and used for another system (see *conversion).*

See also: external system, SEND, RECEIVE, SELF, conversion, USER.

predicate (syntax). The most common kind of premise is a simple predicate, where two expressions are evaluated and the resulting strings or lists are compared. The IS predicate is true if and only if the strings or lists are defined and identical, except for the following special case: If the two values are numbers, they are compared as numbers, not as strings. Thus, 5, 005, 5.0, and " 5 " are all considered to be equal.

The inequality predicates require numeric values for comparison (as always, valid RITA numbers are required).

The IS [NOT] KNOWN predicate tests whether some value has been set or can be evaluated. The IS [NOT] IN predicate tests whether a specific scalar value is a member of a list or a member of any sublist of the list (to any level of nesting).

See *CONTAINS* for an explanation of predicates involving a pattern specification.

Form: value IS [NOT] value
value IS [NOT] KNOWN
value IS [NOT] LESS THAN [OR EQUAL TO] value
value IS [NOT] GREATER THAN [OR EQUAL TO] value
value IS [NOT] IN value
value [CONTAINS] [DOES NOT CONTAIN] pattern specification

See also: premise, value.

premise (syntax). A premise is any clause in the IF part of a rule or goal. Premises may be connected by ANDs or by ORs. AND is the primary connective (i.e., has lower precedence), so the left-hand-side,

color OF ball IS "blue" AND
shape OF block IS "square" OR
name OF external-system IS "TENEX"

would be true if the first clause were true and one or both of the second and third clauses were true. The OR may not be used to connect a THERE IS premise with another premise, since the special definition of the THERE IS would make it meaningless in that context.

See also: predicate, THERE IS.

proto (UNIX command). This command can be given to UNIX before giving the *rita* command. It creates a file that will contain a record of all input and all output on the terminal after the command is given and before a <CTRL/D> is hit. This can be useful for activities such as debugging. The sequence is to give the *proto* command, wait for the next UNIX prompt ("% "), then give the *rita* command. After exiting RITA, hit the <CTRL/D> key (depress the CTRL key and hold it down while hitting the D key). The file created by *proto* is called protocol. (See UNIX file */doc/proto* for documentation.) WARNING: Delete and <CTRL/FS> are not operative inside a *proto.*

Form: PROTO

See also: EXIT, bug, control character.

PUT (action). "Put . . . into" is a predicate clause for use with lists. A value may be "put" only into an attribute (of an object) which is a list. That is, the attribute must not already have a scalar value assigned to it; it must either be a list (which may be empty) or be undefined (not known). If it is a list, value1 is inserted into the list as the last member unless otherwise specified. Value2 must evaluate to a positive integer. (If value2 evaluates to a non-integer number, it is rounded to an integer.) If attribute of object is not known, it is treated as an empty list into which value1 is inserted.

Form: PUT value1 INTO attribute OF object. . .
PUT value1 INTO attribute OF object AS LAST MEMBER. . .
PUT value1 INTO attribute OF object AS FIRST MEMBER. . .

PUT value1 INTO attribute OF object AFTER MEMBER value2. . .

See also: REMOVE, value.

QUIET (command). The *quiet* and *verbose* commands set and unset quiet mode. The default setting is verbose. While in quiet mode, the front end does not issue confirmation messages when loading rules, goals, and objects; does not decompile immediate rules; and does not type the success or failure message when returning to command level. The [replace rule/goal name?] message is suppressed and the replacement always takes place.

NOTE: Files will load (or fload) slightly faster when quiet mode is set; these, like most commands, cannot be in files to be *fload*ed.

See also: VERBOSE.

QUIETLY (syntax). See DEDUCE.

RECEIVE (action). This action is used to receive information from an external system. The port name must match the port name used by the corresponding sends. *User* is a valid port name in this context, but *self* is not. If the port name has not been seen before (or has been killed), a new external system is created with the only initially receivable input being the "%" prompt of the UNIX shell. As with the send action, if the port name is the specially recognized word *user*, no external process is created. Instead, RITA waits for the user to type a complete line on the terminal.

The receive action may be used to receive with or without a test for a matching string (see *pattern),* with or without a time-out (maximum number of seconds to wait for more input to arrive in the buffer), and with or without storing the result as attribute of object.

When RITA is performing a receive without a test for patterns and without a time-out, it sets attribute of object (if any) to a string consisting of the entire input that has arrived from the port. (If the input port is empty, the string will be the null string.)

The receive action without the pattern test and with the time-out (RECEIVE FOR value SECONDS FROM charstr. . .) is performed exactly like the same action without the time-out, except that RITA delays for value seconds before the action is performed. Value must evaluate to a positive number, which will be rounded to an integer.

The action "RECEIVE NEXT pattern specification FROM port name" receives from the port until the specification has been matched, or until the search has definitely failed. Failure can occur under only three conditions:

- An end-of-file is encountered.
- The pattern specification begins with "START FOLLOWED BY pattern", and there is no match for the pattern starting at the first input character.
- A time-out occurs (it can occur only when the "FOR value SECONDS" phrase is used).

When there is a failure, the value of attribute of object (if any) will be set to NOT KNOWN. Any input that has arrived but has not been stored in attribute of object will remain in the input buffer for that port and may be accessed through subsequent receives. When a RECEIVE NEXT pattern is used with a time limit (FOR value SECONDS), the buffer must be found empty value times before the time-out will occur. Each time the buffer is found empty, a 1-second delay will occur before it is checked again.

A RECEIVE from the user port will be received by the RITA agent one line at a time. If a NUL character is received, it will be converted to "~0", since a NUL cannot be part of a RITA string.

Form: RECEIVE FROM port name [AS attribute OF object]
RECEIVE FOR value SECONDS FROM port name [AS attribute OF object]
RECEIVE NEXT pattern specification FROM port name [AS attribute OF object]
RECEIVE NEXT pattern specification FOR value SECONDS FROM charstr [AS attribute OF object]

See also: SEND, port, NUL.

REMOVE (action). REMOVE is a predicate provided to allow members to be removed from a list. REMOVE does the opposite of the PUT action: It modifies the value of a specified attribute, which must be a list. A member or members may be removed from a list according to their position or according to their value.

Remove first member and *remove last member,* together with *remove member value,* remove members according to their position. In this case, value must evaluate to a positive integer no greater than the number of members in the list. If it evaluates to a non-integer number, it will be rounded to the nearest integer. (The phrase SIZE OF can be used to determine the number of members in the list.)

Remove value and *remove first value* remove a member according to its value. They both remove the first member whose value evaluates to value. For example, *remove first "alert" from status-flags of system* will check the list status-flags of the system and remove the first member whose value is "alert" from that list. *Remove every* value *from attribute of object* performs a similar function but does so for every member of the list that has the given value.

Form: REMOVE FIRST MEMBER FROM attribute OF object
REMOVE LAST MEMBER FROM attribute OF object
REMOVE MEMBER value FROM attribute OF object
REMOVE value FROM attribute OF object
REMOVE EVERY value FROM attribute OF object
REMOVE FIRST value FROM attribute OF object

Example: remove "jones" from access-list of remote-file;

See also: PUT.

reserved word. A reserved word is a RITA keyword that may not be used as the name of an attribute, object, rule, or goal. The following are currently the reserved words:

A	IF	OF
AN	IS	SIZE
AND	LINE	THE
CHAR	LINES	THEN
CHARS	MEMBER	WHOSE
FROM	NAMES	

RETURN (action). These actions force the left-hand-side (LHS) monitor to suspend execution and return to command level. The message *Success!* or *Failure* is printed to indicate the type of return, but the difference between the returns has no other effect; the return code is mainly for documentation. The message may be suppressed with the QUIET command. A RETURN FAILURE is automatically provided when all rules have been tried and found to be false.

Form: RETURN SUCCESS
RETURN FAILURE

See also: QUIET.

RHS (right-hand side). See monitor (goal-directed).

rita (UNIX command). The *rita* command is given to UNIX to invoke the RITA processor. Files of rules and objects may be specified in the command line. These files will be loaded when the RITA processes have been initialized, before control is given to the user (see *command level).* As the three major processes (UFE, PARSER, MON) are created, their version dates are printed. The version dates may be suppressed by using the *qrita* (for *quiet)* command. If the user wants a special parser or monitor, (e.g., for testing rule sets designed to run on an 11/40), these may be specified as the first two arguments to the command, preceded by minus signs. When a new version of RITA is being tested, it can be accessed by the *trita* command.

RITA is divided into these three processes in order to overcome UNIX limitations on the size of programs. The separation is usually transparent to the user.

Form: RITA
RITA filename
RITA [–<parser> [–<monitor>]] [<filename> ... <filename>]
QRITA [–<parser> [–<monitor>]] [<filename> ... <filename>]
TRITA [–<parser> [–<monitor>]] [<filename> ... <filename>]

rita.history (file). See WHAT, EXIT, NOFILES.

RULE (command). The basic element of a RITA rule set is the *rule*. Rules have a name, which may be a fairly arbitrary string of characters (but may not be a reserved word), a set of premises, and a set of actions. A *rule* may have the same name as a *goal*, since they are stored separately. The rules are considered to be independent of one another. Although there is an implied overall ordering of them, no rule can explicitly transfer control to another rule. The IF part of a rule consists of a number of premises; all the premises must be true for the rule to be true. If the IF part is true, then the actions are executed sequentially. The premises test various attributes of the data base (see *object*), and the actions can change the data base. In addition, actions can be taken that test and affect the state of external systems, e.g., processes accessing the ARPAnet or dealing with a UNIX data base.

The *rule* command is one way to enter a rule into the RITA system. When the rule is typed in, it is checked for syntactic correctness and then entered into the rule set in RITA. If another rule with the same name already exists, the user is asked whether the old rule should be replaced. Rules may also be read into RITA from UNIX files with the LOAD and FLOAD commands, or as part of the *rita* command to UNIX.

Form: RULE rule-name IF premises THEN actions;

See also: premise, action, monitor, GOAL, object, attribute, value, external system, LOAD, FLOAD, name.

RUN (command). The *run* command is used to start the RITA monitor (specified by a previously issued SET ORDERED or SET UNORDERED). *Run* may be issued more than once. Each time it is issued, the left-hand-side (LHS) monitor starts its scan at the top of the rule list.

See also: command level, SET ORDERED, SET UNORDERED, monitor, CONTINUE, RUN RULE, TRACE, RETURN, STOP AT, FAILURE.

RUN RULE (command). This is another form of the *immediate rule* command, the difference being that the user is referring to a named rule (but not goal) that is already in the system. The same operation applies to the named rule as to the unnamed (immediate) variety. (Like other commands, *run rule* may not be used as the action of a rule.)

Form: RUN RULE rule-name;

See also: immediate rule, RUN.

scalar. A scalar is also known as a string.

See also: data type, string.

SCRATCH (command). The *scratch* command is identical to the command *delete all rules and delete all goals and delete all objects.*

See also: DELETE.

SELF (pre-initialized port). New rules and objects may be added to a rule set by constructing them through the actions of rules and sending them to the current RITA using the SELF pipe. This pipe is read just before RITA reenters command mode, and the actions specified (e.g., adding rules, setting traces, etc.) are taken then. The SELF pipe may also be used for explicitly deleting rules, deciding which file to load next, exiting conditionally, and so on.

Unfortunately, there are many restrictions on the use of the send-to-self feature: Reading the self strings operates like loading a file and is therefore subject to the quadruple-nesting limit (see *LOAD)*; like typed input, nothing should be put into the self channel after a load command; RITA must return to command level before it can start reading the self strings; and an attempt to send more than 4098 characters to self without such a return will cause the system to go into an infinite wait.

See also: external system, SEND, LOAD, command level, RUN, RETURN, CONTINUE.

SEND (action). The value must evaluate to a string or a list (NOT KNOWN values cause an error). It is transmitted to the named port; if the port name has not been seen before in a *send* or *receive,* a new external system by that name is created, if there are enough resources to support another external system. (Currently, up to three such systems may run simultaneously.) This system is initialized to be a UNIX shell, and this shell will receive the string being sent. Thus, the first string sent to a new external system is generally the shell command to run the appropriate program.

Two external system names are treated very specially: *user* and *self.* Neither of these is really an external system, and neither causes the creation of a shell or any other processes. Sending something to the user causes it to be printed on the standard output (the user's terminal). The self pipe is discussed under *SELF.*

The value is sent in its literal form, followed by a newline, except that characters preceded by an up-arrow (↑) are converted into special ASCII codes at parse time, and characters preceded by a tilde (~) are converted into special control functions which are performed on this port. The specific conversions are covered under *conversions.*

The name chosen for an external system cannot be changed after it has been assigned. For documentation purposes, it should probably be the name of the system that will be most heavily talked to along that I/O channel; for example, send "net" to ARPAnet and send "who" to shell;

A special case of the send action concerns the SEND. . .TO FILE option. In this case, no external (or self or user) system is involved. Instead, the string is appended to the specified file (which is created if it does not already exist). Many of the "~" codes have no meaning in this case and are just copied into the string without any special control functions being executed.

Form: SEND value TO [FILE] port-name

See also: port, RECEIVE, SELF, conversion, DISPLAY [TO FILE], NUL.

SET ORDERED (command). This command specifies that the ordered left-hand-side (LHS) monitor is to be used instead of the default unordered monitor (see *monitor*). The setting may be changed any time a command may be issued (i.e., after an asterisk prompt).

See also: SET UNORDERED, monitor.

SET ... TO (action). In the form *SET attribute OF object TO value,* the old value of the attribute of object is replaced with the one specified by the value. If there was no such attribute previously, one is created.

In the second form of the *set* action shown below, the value of the attribute of object being set is replaced by a list which contains the values of the named attribute of all objects that have an attribute which satisfies the predicate. For example, *Set the active-sites of network to the name of every remote-site whose status is "active"* will put the name of every remote-site whose status is "active" into a list called active-sites, which is an attribute of network. If there are no remote-sites that satisfy the predicate, the list will be empty.

Form: SET attribute OF object TO value
SET attribute OF object TO attribute
OF EVERY object WHOSE attribute predicate

See also: predicate.

SET TO NOT KNOWN (action). Any value of a specified attribute of the object is forgotten and is treated subsequently as if it had never been known before. If the attribute was not mentioned prior to execution of this action (and thus was unknown anyway), this action will cause the fact that the object has such an attribute (even though still unknown) to be displayed when decompiled. This action is not legal in a goal rule.

Form: SET attribute OF object TO NOT KNOWN

See also: deduce.

SET TRACE (command). This debugging command causes RITA to report its actions in more detail. The higher the integer, the more detail is printed. Currently, 4 is the maximum meaningful integer; 0 specifies the normal mode of operation. If n is greater than 0, output will be produced by immediate actions and rules as well as by the monitor.

The messages currently produced include the following: *trace level 1* prints all the rules and goals being tested; *trace level 4* includes, among other things, announcements of each rule that fires and a list of its actions as they are about to be executed.

Form: SET TRACE integer;

See also: TRACE.

SET UNORDERED (command). This command specifies that the cyclic LHS monitor (see *monitor*) is to be used. Since it is the default, this command need only be given at some point after the RUN ORDERED command has been

given. The setting may be changed any time a command may be issued (i.e., after an asterisk prompt).

See also: SET ORDERED, monitor.

SHELL (command). The *shell* command temporarily puts the user into a UNIX shell (e.g., for checking directories or for sending a message). RITA is left suspended unless the user makes the tragic mistake of typing <CTRL/FS>, in which case it goes away with three core dumps (one for each process). To return to the RITA system—and the user *must not* forget to do so—he needs only type a <CTRL/D> to the (sub)shell. Typing more than one <CTRL/D> will get him out of RITA as well.

Form: SHELL;

SIZE (syntax). The SIZE operator returns the number of elements in the specified list. It can be used in a premise, e.g.,

IF SIZE OF foo OF baz IS LESS THAN 5 THEN ...

or in an action, e.g.,

... THEN SET count OF prompts TO SIZE OF prompts OF system ...

and is legal wherever a value may be used. If the value whose size it is taking is not a list, a run-time error is displayed.

Form: SIZE OF list

See also: length (of a string), sused (amount of memory used).

STOP AT. See TRACE.

string. A string consists of zero or more characters between quotation marks. Note that upper and lower case are distinguished in quoted strings. A string is also known as a scalar.

See also: value, conversion, data type, name.

subtraction. See arithmetic.

sused (built-in function). The function *sused* returns the amount of memory (in K words) used so far. The maximum memory available to a process under UNIX is 28K (K = 1024). The PDP 11/45 (or 11/70) version of RITA currently returns 6 in an empty RITA (leaving room for 22K expansion of the user program), and the PDP 11/40 version returns 22 (room for 6K expansion). After reaching the 28K point, RITA attempts to keep the user in business by swapping rules dynamically to disk and reclaiming space occupied by unused strings. The difference between the 11/40 and the 11/45 is that on the 11/45, instructions and data may be addressed separately, so that in the best case the 11/45 provides twice as much address space.

Form: SUSED( )

syntax. See App. A for a complete syntax chart showing legal RITA rule descriptions, object descriptions, and command syntax.

THE (syntax). This is one of several optional words which may be used to improve the readability of rules. It is thrown away/or and ignored whenever it is encountered by the system.

See also: article.

THEN (syntax). The keyword THEN is used to introduce the actions of a rule or goal.

See also: rule, goal, action, immediate action, immediate rule, IF.

THERE IS (syntax). THERE IS is a premise used for selecting a single object from a group of objects of the same type. For example, if the data base contains a number of records, one could be selected with the clause, *THERE IS A record WHOSE id-field IS "vacant".* The monitor goes to considerable effort to find out whether there is such an object. It will try each object of type "record" in turn; if it finds one with a matching id-field, it will then test the remaining premises in the IF part of the rule. If all premises are true, then the rule is true. If any premise is false, the monitor comes back to the THERE IS clause and tries the next record until it either finds a record that makes all subsequent premises true or runs out of records (in which case, the rule is false). If there are a number of THERE IS clauses in the rule, a good deal of testing can take place. For example, three THERE IS clauses each binding an object "foo" could generate 125 binding attempts if there were 5 objects of type "foo" in the data base.

Form: THERE IS [NOT] objectname
THERE IS [NOT] [A][AN] objectname WHOSE attribute IS value [AND WHOSE ...]
THERE IS [NOT] objectname WHOSE attribute IS LESS THAN [OR EQUAL TO] value [AND WHOSE ...]
THERE IS [NOT] objectname WHOSE attribute IS GREATER THAN [OR EQUAL TO] value [AND WHOSE ...]
THERE IS [NOT] objectname WHOSE attribute IS IN value [AND WHOSE ...]
THERE IS [NOT] objectname WHOSE attribute [CONTAINS] [DOES NOT CONTAIN] pattern specification

See also: premise, object, CONTAINS.

TRACE (action). The four *trace* commands—TRACE, UNTRACE, STOP, UNSTOP—set and unset flags associated with various RITA items (rules, goals, objects, and attributes). The syntax to select the desired items is as follows: *ALL RULES, ALL GOALS, ALL OBJECTS, RULE rulename, and GOAL goalname* are all self-explanatory (*all* by itself refers to the combination of the first three groups, i.e., everything); *OBJECT objectname* refers to all objects of class *objectname*, but if a local label or explicit binding (the <> feature) is used, it refers only to that particular object; *ALL RULES THAT test/set*

attribute OF object refers to a subset of rules or goals that *tests* an attribute of object if the attribute/object pair is mentioned anywhere in its premise, and *sets* it if it is mentioned in a SET, CREATE, RECEIVE, PUT, or REMOVE clause in any of its actions; and *attribute OF object* refers to the attribute of one or every object of type *object*, depending on whether or not the object reference can be bound.

In any event, every rule, goal, object, and attribute has two flags: a trace flag and a stop flag. For rules and goals, being traced means that announcement is made every time the rule/goal is fired; being stopped means that every time the rule/goal is about to fire (i.e., is found true but before any of its actions are executed), execution is suspended and control is returned to command level. If a *continue* command is issued, the rule will then fire if it is still true and will then continue. (The *run* command, however, will stop at the rule again unless it has been unstopped or is no longer true. The rule may be false if the user has changed a value before continuing.) For objects and attributes, tracing means that the specified item's name and value are announced every time the value changes (in the case of an object, they are announced any time any of its attribute's values change); stopping produces similar results, except that execution is suspended before the change of value takes place. A *continue* command will perform the assignment and finish executing any other actions in the current rule.

Thus, these commands give the user a powerful set of options for observing and intervening in the operation of his RITA programs.

Form: TRACE RULE rulename
TRACE GOAL goalname
TRACE ALL RULES
TRACE ALL RULES THAT SET attribute OF object
TRACE ALL RULES THAT TEST attribute OF object
TRACE all objects
TRACE OBJECT objectname
TRACE attribute of object
UNTRACE ...
STOP AT ...
UNSTOP ...

See also: SET TRACE.

"TRUE" (value). Some built-in functions such as *isnum* and *islist* return the literal string "TRUE" if their conditions are satisfied. Note that the string is in upper-case letters.

See also: "FALSE".

uc (upper case) (built-in function). The argument should evaluate to a string. If so, the returned value is the same string with all lower-case letters shifted to upper case. If not, a run-time error message is given.

Form: UC(value)

ufe (user front end). See rita.

ufe.input. See edit.

ufe.output. See edit.

UNTRACE. See TRACE.

UNSTOP. See TRACE.

upper case. See uc.

USER (port). This is the name of a pre-initialized port that may be used to send data to the controlling terminal of a process or to receive data from it. If the user is at a terminal, information sent to the user port with the SEND command will appear on the user's terminal, and characters typed at the terminal can be read with a RECEIVE specifying the user port. DISPLAY also sends information to the USER port.

See also: port, SELF, SEND, RECEIVE, conversion.

value (syntax). A RITA value can be either a string of characters, an ordered list of values, or unknown.

A string is a sequence of characters or a RITA number. (See *data type.)* To enter a string value, the user surrounds the characters by double quotes ("); to include a double quote within the string, he precedes it with an up-arrow (as in "quote = ↑ ""). Values obtained through a receive action may contain all ASCII characters except NUL; strings typed in as part of a rule or object may contain non-printing characters by making use of the ↑-conversions described under *conversion.*

A list is entered as a left parenthesis, a possibly null number of values (strings or lists) separated by commas, and a right parenthesis. Lists used as values (i.e., in rules) may contain any legal values within them (for example, ("foo", ( ), color of ball, 1 + 2, size of b of x)), but literal lists (used in the object description syntax) may contain only strings and other literal lists (for example, the last three members in the above example would be illegal in a literal list). That is, no value that needs to be evaluated is allowed in the object description syntax. (See *object description.)* If an object is to be created with evaluated attributes, the CREATE syntax, which is also more readable, should be used. Note that a list may include sublists as elements and that duplicate entries are allowed in a list.

Unknown values are identical to values that do not exist (e.g., a ball with no color attribute and a ball with a color attribute whose value is NOT KNOWN are treated identically). An unknown value is entered by using the phrase *not known.*

Valid means of specifying a value within a RITA rule set are shown on the wall chart in App. A within the box labeled "a VALUE can be."

VERBOSE (command). The *quiet* and *verbose* commands set and unset quiet mode. The default setting is *verbose.* While in quiet mode, the front end does not issue confirmation messages when loading rules, goals, and objects; does

not decompile immediate rules; and does not type the *Success* or *Failure* message when returning to command level. The [replace rule/goal name?] message is suppressed and the replacement always takes place.

Files will load (or fload) slightly faster when quiet mode is set; these, like most commands, cannot be in files to be *fload*ed, but they may be in files that were compiled.

See also: QUIET.

WHAT (command). RITA keeps a running record of all the major events that take place as it operates unless the *nofiles* command is issued. The explanation system utilizes this record to explain RITA's behavior to the user. (The file, called *rita.history*, is useful for postmortem analysis). For user convenience, this record is broken down into epochs delimited by issuances of the *run* command and broken down further into events. Event 0 includes everything before the first *run* command (generally a series of loaded rules and objects); every time a rule or goal fires, the event counter is incremented. A range of events may be referred to as follows (n and m are unsigned integers): 'n-m' refers to events n through m inclusive; 'n' refers to event n; 'n-' refers to events n to the present (last event); '-n' refers to events from the last *run* command to event n inclusive; '-' refers to the events since the last *run* command; and no range specification is equivalent to the range '-'. There is currently no way to refer to the last event; using a number too large will result in a harmless *out of range* message.

The *what* command simply displays a readable version of what went on. The level of detail will eventually be selectable; currently, the command displays the details of all additions and changes to the data base (though not deletions), announces any I/O that took place (though not the actual messages communicated), and lists the rules and goals that fired and the (deductive) questions that were asked the user.

At present, the printout from a *what* command will roll off the screen as fast as it is typed; the printout may be halted by typing a <DEL>.

Form: WHAT range?

WHY (response to deduction question). Responding 'why' to a question asked by the deductive monitor results in a listing of the immediate subgoal and rule that were responsible for the question. Repeated whys follow the chain of logic created by the backward-chaining process, up to the initial goal. When the top of the decision tree is reached (i.e., the reason for needing a datum is the action DEDUCE attribute OF object), RITA replies "That's what I was supposed to deduce." The *why* response may be used only during a deduction. It is not available at the command level or within rules or goals.

See also: DEDUCE, deduction.

Appendix A

SUMMARY OF RITA COMMANDS AND SYNTAX

Appendix A is a wall chart illustrating the commands and syntax within the RITA system. This chart is contained in the pocket on the inside back cover of this report.

Additional copies of this chart may be obtained from the Publications Department, The Rand Corporation, 1700 Main Street, Santa Monica, Calif. 90406.

Appendix B

A RITA RULE SET FOR FILING MAIL

This appendix describes the creation and operation of an illustrative user agent whose function is to file electronic computer network mail. It is assumed that the user has some knowledge of the format of network mail, but this is not essential.

The agent is designed to go through an "in box" file of network mail and file the messages in various data sets. Incoming messages are filed according to sender; outgoing messages are filed according to recipient(s). The agent has in its data base a set of known persons with whom the user corresponds. (In some cases, the "person" is actually a group, e.g., the MSGGROUP on the ARPAnet; these groups can be treated as individual persons.) Each such person will have a formal network account name and an associated file name into which messages to or from him should be filed. (Messages from different persons can thus all be routed to one common file, if desired.) The agent should look at each message in turn within a given mailbox (ignoring deleted messages that have not yet been expunged). For each message, it should look within the *To:* and *From:* fields (but not the *Cc:* field) for names of known persons; messages for each known person found should be filed in the file designated for that person. The user should be informed of all such actions, and the original mailbox should not be destroyed in the process. (In order to keep the discussion of this agent brief, we ignore such potential problems as that of placing multiple copies of a message in the same file (i.e., once for each correspondent).)

Figure B-1 illustrates sample interactions with MS,* a newly-developed UNIX-based message system that is available at Rand and can be used by RITA. Annotations to the transcript in Fig. B-1 are shown in braces.

Figure B-2 presents a diagram of the logic underlying this user agent. The capabilities of the MS system which can be applied to the task are taken into account. The flow chart boxes have been labeled with phrases which ultimately can be used to represent "states" of the RITA agent. Such agents can be thought of as being in one of a number of states, where certain production rules apply in each state.

From the logic flow in Fig. B-2, it is straightforward to encode the behavior of an agent as a set of RITA production rules. First, a set of known-person objects with associated attributes and a similar object called *me* will be created. Note that each known-person has a corresponding file name into which messages should be stored; the file name associated with *me* is the one into which miscellaneous messages will be filed. The data objects, expressed in RITA syntax, are shown in Fig. B-3.

In addition to the data objects in Fig. B-3, a few others are useful in expressing the logic of a mail-filer agent: an "agent" which has an associated state; an "ms" object representing the MS system, its prompt character(s), its responses, etc.; a "message" object recording information about the to- and from-fields of the current

*See David Crocker, *User-Level Functions in MS: A Network-Oriented Message System for Personal Computing*, The Rand Corportion, R-2134-ARPA (to be published).

```
% ms newmail        {"ms" is the unix command to invoke the
                     mail system;  the optional argument "newmail"
                     tells the system which mailbox to use}
MS: 22-Dec-76
13 messages in mailbox.

-> scan all         {"->" is the MS prompt; look at all the
                     message headers.  The symbols at the left
                     margin by message #3 indicate that message
                     has been deleted, but not expunged.}
     1<= (1243)
     2   (1169) 29 Oct 76 jsz              Virtual Terminal UNIX
*[-  3    (8Ø5) 29 Oct 76 anderson         Re: [Virtual terminals]]
     4   (2224) 29 OCT 76 NELC3Ø3Ø         RITA Agents
     5    (929) 3Ø OCT 76 MSGGROUP         MSGGROUP# 429 Add RCT@CCA
     6    (521) 29 Oct 76 VITTAL           How to get the draft RFC
     7   (1Ø76) 29 Oct 76 VITTAL           Draft of RFC on new mail
     8   (1374) 31 OCT 76 FARBER           Your draft on message sys
     9   (1149) 1 NOV 76  WALKER           UNIX Meeting 3 Nov
    1Ø    (319) 1 Nov 76  MYER             Inadvertent Messages
    11   (1792)  1 Nov 76 greep            Re: [FILES CREATED IN MS
    12    (683)  1 Nov 76 greep            Re: [FILES CREATED IN MS
    13    (29Ø)  1 Nov 76 grm              ACCAT R & D

-> show 2
(Message 2, 1169 bytes)
Date: 29 Oct 1976 at 1317-PDT
From: jsz
Subject: Virtual Terminal UNIX
To: jim
cc: rha, weiner, jsz
```

Figure B-1

```
Jim,

        In response to your inquiry concerning virtual terminal
UNIX ...   I would estimate that this work will be started in
about a month, and might take a month or two.

                                        Steve

-------

-> show 2 -c to from       {the "-c" flag tells the show command
                            to only display the named components
                            of the message(s)}
(Message 2, 1169 bytes)
To: jim
From: jsz

-> copy 2 > -f m.jim       {places a copy of message #2 in file
                            "m.jim";  the "-f" flag indicates that
                            the next argument is a file name}

-> show 3 -c to from
Message 3 is discarded.    {from this, we learn what MS says
                            when we access a discarded (but not
                            expunged) message}

-> show 14 -c to from
Message 14 does not exist!    {this is what MS says when we access
                               a message beyond the end of the
                               list of messages;  by testing for
                               this statement, we can tell when
                               all messages have been sequentially
                               processed}
-> quit
%                          {we have returned to the UNIX shell}
```

Figure B-1--continued

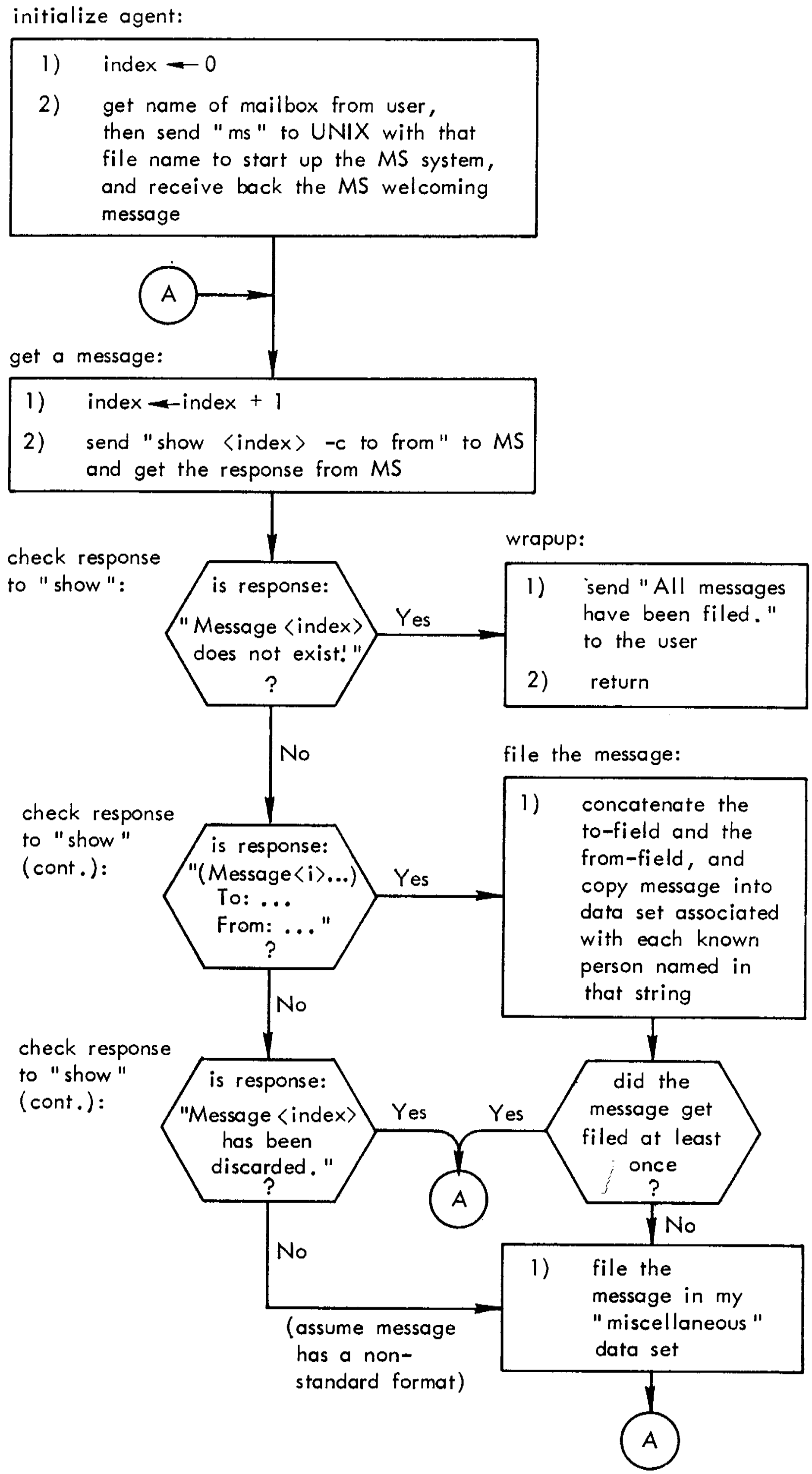
initialize agent:
1) index ← 0
2) get name of mailbox from user, then send "ms" to UNIX with that file name to start up the MS system, and receive back the MS welcoming message
A
get a message:
1) index ← index + 1
2) send "show <index> -c to from" to MS and get the response from MS
check response to "show":
is response: " Message <index> does not exist! " ?
Yes
wrapup:
1) send "All messages have been filed." to the user
2) return
No
check response to "show" (cont.):
is response: "(Message<i>...) To: ... From: ... " ?
Yes
file the message:
1) concatenate the to-field and the from-field, and copy message into data set associated with each known person named in that string
No
check response to "show" (cont.):
is response: "Message <index> has been discarded." ?
Yes
A
Yes
did the message get filed at least once ?
No
1) file the message in my "miscellaneous" data set
No
(assume message has a non-standard format)
A

Figure B-2

```
object me:            name is "Robert Anderson",
                      address is "anderson",
                      msg-file is "m.misc";

object known-person:  name is "Steve Walker",
                      address is "walker",
                      msg-file is "m.walker";

object known-person:  name is "Cmdr. Floyd Hollister",
                      address is "fhollister",
                      msg-file is "m.hollister";

object known-person:  name is "Cmdr. Cliff Rose",
                      address is "rose",
                      msg-file is "m.nelc";

object known-person:  name is "NELC people",
                      address is "nelc3030",
                      msg-file is "m.nelc";

object known-person:  name is "Dave Farber",
                      address is "farber",
                      msg-file is "m.farber";

object known-person:  name is "CAHCOM committee",
                      address is "cahcom",
                      msg-file is "m.cahcom";

object known-person:  name is "MSGGROUP",
                      address is "msggroup",
                      msg-file is "m.msggroup";

object known-person:  name is "Gary Martins",
                      address is "grm",
                      msg-file is "m.nelc";
```

Figure B-3

message; and an object called "processed-persons" which keeps as an attribute a list of the names of known persons already used in filing a message, as a means of bookkeeping during iteration through all known persons. These additional data objects are shown in Fig. B-4.

```
object agent:          state is "initialize agent";


object ms:             response is "",

                       name is "ms",    [UNIX command to invoke MS]

                       prompt is "%j-> ";      ["%j" is carr. ret.]


object message:        address-field is "",

                       status is "unfiled";


object processed-persons:

                       list is ();      [used for bookkeeping, to

                                         remember which persons a

                                         message has already been

                                         filed under]
```

Figure B-4

Most of the initialization of attribute values shown in Fig. B-4 is unnecessary for operation of the RITA agent; if an attribute is not declared as part of the definition of a data object, it will be dynamically created when it is first used. The initialization is shown in Fig. B-4 to provide documentation of the various attributes used in the agent's rules.

The agent's logic can now be described in nine RITA rules, using the vocabulary defined in the data objects of Figs. B-3 and B-4. The rules are shown in Fig. B-5.

The rules in Fig. B-5 are self-explanatory; however, the following additional details may be helpful:

1. The string "↑j" represents the special control character, carriage return (see *conversion* in Sec. III).
2. Text within braces describes a pattern to be matched by a string. Several options are available within a pattern description; see *pattern specification* in Sec. III.
3. The built-in functions used within the rules are:

   - concat(s1, s2, . . ., sn): returns strings s1,. . .,sn concatenated together.
   - lc(s): returns lower-case version of string s.

```
set ordered;  [tell RITA to treat this as an ordered rule set]
quiet;        [the RITA system should be terse in informing
               the user re. various details]
RULE 1: [HOW TO INITIALIZE THE AGENT]

  IF:   the state of the agent is "initialize agent"

  THEN: set the index of the message to 0
        and send concat(
        "Please type file name containing messages to be filed%j",
        " (or hit RETURN for default mailbox)...")
                to user
        and receive next {anything 'file-name' followed by "%j"}
                from user
        and send concat(name of ms, " ", 'file-name') to ms-port
        and receive next {prompt of ms} from ms-port
        and set the state of the agent to "get a message";

RULE 2: [HOW TO GET A NEW MESSAGE FROM 'MS']

  IF:   the state of the agent is "get a message"

  THEN: set the index of the message to index of message + 1
        and send concat("show ", index of message, " -c to from")
                to ms-port
        and receive next {prompt of ms}
                from ms-port as the response of ms
        and set the status of the message to "unfiled"
        and set the list of processed-persons to ()
        and set the state of the agent to
                "check response to 'show'";
```

Figure B-5

```
RULE 3: [HOW TO TELL IF ALL MESSAGES HAVE BEEN PROCESSED]

   IF:    the state of the agent is "check response to 'show'"
          and the response of ms contains {"Message " followed by
              some in "0123456789" followed by " does not exist"}

   THEN:  send "All messages have been filed." to the user
          and return success;

RULE 4: [HOW TO EXTRACT THE NEEDED INFORMATION FROM A MESSAGE]

   IF:    the state of the agent is "check response to 'show'"
          and the response of ms contains {"(Message " followed by
              anything followed by "%jTo: " followed by
              anything 'to-string' followed by "%jFrom: "
              followed by anything 'from-string' followed by
              "%j" followed by anything followed by prompt of ms}

   THEN:  set the address-field of the message to
                  lc(concat( 'to-string', " ", 'from-string'))
          and set the state of the agent to "file the message";

RULE 5: [WHAT TO DO IF THE CURRENT MESSAGE HAS BEEN DISCARDED]

   IF:    the state of the agent is "check response to 'show'"
          and the response of ms contains {"Message " followed by
              anything followed by " is discarded." followed by
              anything followed by prompt of ms}

   THEN:  set the state of the agent to "get a message";
```

Figure B-5—continued

```
RULE 6: [WHAT TO DO IF THE CURRENT MSG HAS A NON-STANDARD HEADER]

  IF:    the state of the agent is "check response to 'show'"
         [and since above rules didn't fire, assume that message
          is non-standard format, so file as if no known-person
          were mentioned]

  THEN:  set the address-field of the message to ""
         and set the state of the agent to "file the message";

RULE 7: [HOW TO FILE A MESSAGE TO/FROM A KNOWN PERSON]

  IF:    the state of the agent is "file the message"
         and there is a known-person whose address is not in the
                 list of processed-persons
         and the address-field of the message contains
            {address of known-person}

  THEN:  send concat("copy ", index of message, " > -f ",
                    msg-file of known-person) to ms-port
         and receive next {prompt of ms} from ms-port
         and send concat("Message # ", index of message,
                   " placed in file ", msg-file of known-person)
                      to user
         and put the address of the known-person into
                 the list of processed-persons
         and set the status of the message to "filed";

RULE 8: [WHAT TO DO IF NO EXPLICIT FILING INSTRUCTION HAS BEEN
         SUCCESSFUL:  FILE UNDER MY MISCELLANEOUS]

  IF:    the state of the agent is "file the message"
         and the status of the message is not "filed"
```

Figure B-5—continued

```
THEN: send concat("copy ", index of message, " > -f ",
                  msq-file of me) to ms-port
      and receive next {prompt of ms} from ms-port
      and send concat("Message # ", index of message,
                  " placed in file ", msq-file of me) to user
      and set the status of the message to "filed";

RULE 9: [WHAT TO DO AFTER THE CURRENT MESSAGE HAS BEEN FILED]

  IF:   the state of the agent is "file the message"
        and the status of the message is "filed"

  THEN: set the state of the agent to "get a message";

run;            [after loading in the rules, RITA should begin
                 execution of this agent]
```

Figure B-6 records a session at which the agent is invoked. Again, annotations are shown in braces.

```
% rita mail-filer                  [invoke RITA, loading mail-filer]

There are 52 processes in use (61% loading) and no other ritas
                                                          running.

UFE:     11 Nov 76                 [RITA consists of three processes:
PARSER:  9 Dec 76                   User Front End (UFE), a Parser,
MON:     11 Dec 76                  and a Monitor (MON);  this
                                    indicates which versions are
                                    in use]

mail-filer:                        [RITA provides feedback while
                                    loading data and rules]
```

Figure B-6

```
Please type file name containing messages to be filed
 (or hit RETURN for default mailbox)...

newmail                          [this is the user's response to
                                  the agent's question]

Message # 1 placed in file m.misc
Message # 2 placed in file m.misc       [feedback to the user
Message # 4 placed in file m.nelc        programmed into the
Message # 5 placed in file m.msggroup    agent's behavior in
Message # 6 placed in file m.cahcom      rules #7-8]
Message # 7 placed in file m.cahcom
Message # 8 placed in file m.farber
Message # 9 placed in file m.walker
Message # 9 placed in file m.farber
Message # 10 placed in file m.msggroup
Message # 11 placed in file m.misc
Message # 12 placed in file m.misc
Message # 13 placed in file m.nelc
All messages have been filed.

* exit;                          [user exits RITA, returning to
                                  UNIX command level]
exiting.

%
```

Figure B-6—continued

We have shown this illustrative agent as an interactive one, which requests a file name from the user and displays feedback during its operation. It is also possible, with minor modifications, to make this agent operate autonomously, waking up periodically to process messages, etc., without requiring user interaction. Such an agent can leave messages for the user, reporting on what has been done.

RAND/R-1808-ARPA